Augmenting IDEs with Runtime Information for Software Maintenance

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern
vorgelegt von

David Röthlisberger

von Langnau (BE)

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Der Dekan:

Bern, 04.06.2010

Prof. Dr. U. Feller

This dissertation is available as a free download from http://scg.unibe.ch/

Published by David Röthlisberger, Switzerland
ISBN 978-1-4457-6026-1
First Edition, May 2010

Acknowledgments

I was only able to complete this dissertation thanks to the kind help of people who gave me advice, hints, ideas, or encouragement.

Most notably, I want to thank Oscar Nierstrasz for giving me the opportunity to work on this dissertation at the Software Composition Group. Without his continuous support, his professional advice, and his encouraging feedback on my work it would not have been possible to finish this dissertation.

I am also grateful to Harald Gall for being the external reviewer of this dissertation, for carefully reading and evaluating this document, and for coming to Bern to join the jury of the PhD defense. I also thank Torsten Braun for accepting to chair the examination and the PhD defense.

I want to thank particularly Stéphane Ducasse who introduced me to the Software Composition Group, motivated me to pursue a PhD, and gave me countless ideas and visions that highly influenced this work. Without the inspiring discussions and email conversations we had, without Stef's enthusiasm, his priceless advice and his infinite, inexhaustible passion for the things he is doing, this dissertation would not be the same.

Special thanks also goes to Orla Greevy who taught me how to write papers and conduct research and who created a warm working environment when I started with this dissertation. I also thank Tudor Gîrba for all the countless and intensive discussions which often sharpened my awareness for important aspects of research. Moreover, I thank Marcus Denker who supervised my Master's thesis and back then encouraged me to pursue a PhD by introducing me to the world of research.

I specifically also thank people that provided appreciated feedback on drafts of this dissertation: Oscar Nierstrasz, Orla Greevy, Tudor Gîrba, Niko Schwarz, Andreas Fischer, Andreas Thomet, Sabine Benz.

Many thanks go to the present and former Software Composition Group members: Adrian Kuhn for his inspiring nature, his imaginative

appeal, and for the room sharing in Vancouver; Adrian Lienhard for his even-tempered, unagitated yet impressing, assiduous work; Fabrizio Perin for all the funny discussions, the splendid Italian lessons, and the entertaining adventures in Lille; Lukas Renggli for his unrivaled Smalltalk skills and his quality awareness; Jorge Ressia for his unique sense of humor; Niko Schwarz for the funny exercise sessions and his unadulterated belief in research; Toon Verwaest for his inspiring enthusiasm; Erwaan Wernli for his feature-length presentations; Marcus Denker for his socializing spirit and his unshakable belief in the good of Smalltalk; Markus Gälli for his cheerful mindset; Tudor Gîrba for his helpful attitude and his willingness to share his knowledge; Orla Greevy for her amiable nature, the pleasant co-operation, and the good time we shared in the same office.

Thanks also to Therese Schmid and Iris Keller for their excellent support with the administrative chores. I particularly also thank Marcel Härry for his studious, persevering work on *Senseo* and Orla Greevy for her contributions to *FeatureEnv*!

I also thank the external people I collaborated with, most notably Danilo Ansaloni, Alexandre Bergel, Walter Binder, Simon Denier, Philippe Moret, Damien Pollet, Romain Robbes, and Alex Villazón. All of them contributed to this dissertation. I particularly thank Alexandre Bergel for inviting me to Santiago de Chile and for the nice time we had there.

Furthermore, I thank various persons I met during the work on this dissertation at conferences, during meetings or research visits: Hani Abdeen, Marco D'Ambros, Alberto Bacchelli, Jérémy Barbay, Gwenael Casaccio, Johan Fabry, Thomas Fritz, Sonia Haiduc, Lile Hattori, Michele Lanza, Jannik Laval, Mircea Lungu, Fernando Olivero, Daniel Ratiu, Eric Tanter, and Richard Wettel.

Special thanks go to Anina Bachem, Michelle Bauer, Katharina Ledermann, Rachel Martins, and Petra Schilling who were somehow, for one or the other reason part of the process.

I thank my family, Heidi, Andreas, Erwin, Mirjam, and Stefan. Above all, I thank Therese for her heart of gold, her greatness of mind, and her unconditional support.

My words fly up, my thoughts remain below. Words without thoughts never to heaven go.
— William Shakespeare

Abstract

Object-oriented language features such as inheritance, abstract types, late-binding, or polymorphism lead to distributed and scattered code, rendering a software system hard to understand and maintain. The integrated development environment (IDE), the primary tool used by developers to maintain software systems, usually purely operates on static source code and does not reveal dynamic relationships between distributed source artifacts, which makes it difficult for developers to understand and navigate software systems.

Another shortcoming of today's IDEs is the large amount of information with which they typically overwhelm developers. Large software systems encompass several thousand source artifacts such as classes and methods. These static artifacts are presented by IDEs in views such as trees or source editors. To gain an understanding of a system, developers have to open many such views, which leads to a workspace cluttered with different windows or tabs. Navigating through the code or maintaining a working context is thus difficult for developers working on large software systems.

In this dissertation we address the question how to augment IDEs with dynamic information to better navigate scattered code while at the same time not overwhelming developers with even more information in the IDE views. We claim that by first reducing the amount of information developers have to deal with, we are subsequently able to embed dynamic information in the familiar source perspectives of IDEs to better comprehend and navigate large software spaces. We propose means to reduce or mitigate the information by highlighting relevant source elements (*HeatMaps*), by explicitly representing working context (*SmartGroups*), and by automatically housekeeping the workspace in the IDE (*AutumnLeaves*). We then improve navigation of scattered code by explicitly representing dynamic collaboration (*Hermion*, *Senseo*, *CollView*) and software features (*FeatureEnv*) in the static source perspectives of

IDEs. We validate our claim by conducting empirical experiments with developers and by analyzing recorded development sessions.

Contents

List of Figures **xiii**

List of Tables **xix**

1 Introduction **1**

1.1 Problems of Traditional IDEs 1

1.1.1 Development Activities 2

1.1.2 Problem Identification 4

1.1.3 Taxonomy of IDE Problems and Development Activities 9

1.2 Proposal: Tackling Overloaded Views and Integrating Dynamic Information in IDEs 12

1.2.1 Mitigating Information Overload in IDEs 12

1.2.2 Enhancing IDEs with Dynamic Information 14

1.2.3 Summary 16

1.3 Contributions 19

1.4 Structure of the Dissertation 20

2 State of the Art **23**

2.1 Development Environments 23

2.1.1 Program Analysis and Sophisticated Information Presentation 24

2.1.2 Source History Analysis 32

2.1.3 Developer Activity Analysis 35

2.1.4 Debugging, Profiling 41

2.1.5 Querying 45
2.1.6 Conclusions 49
2.2 Software Analysis and Visualization 50
2.2.1 Means to Present Static or Historical Information . 51
2.2.2 Dynamic Analysis 54
2.2.3 Summary 60
2.3 Conclusions 60

I Mitigating Information Overload in IDEs 63

3 HeatMaps – A Navigational Aid 67
3.1 Introduction 67
3.1.1 Positioning *HeatMaps* 67
3.1.2 Introduction to *HeatMaps* 68
3.2 Information Overflow and Overload in IDEs 70
3.2.1 Motivating Use Case 70
3.2.2 Development Driven Information 71
3.3 HeatMaps 72
3.4 Validation 76
3.4.1 Efficiency of HeatMaps 77
3.4.2 Accuracy of HeatMaps 77
3.4.3 User feedback 83
3.5 Related Work and Discussion 84
3.5.1 Related Work 84
3.5.2 Discussion 86
3.6 Summary of the Chapter 88

4 SmartGroups – Representing Context in IDEs 91
4.1 Introduction 91
4.1.1 Positioning *SmartGroups* 91
4.1.2 Introduction to *SmartGroups* 92
4.2 Software Space Navigation Issues 94

4.3 Existing Approaches 96
4.4 *SmartGroups* in a Nutshell 98
4.4.1 Automatic Smart Groups 98
4.4.2 Manual Smart Groups 107
4.4.3 Query Results as Smart Groups 107
4.4.4 Integration of the *SmartGroups* View 108
4.5 Validation 109
4.5.1 Correctness of *SmartGroups* 109
4.5.2 User Feedback 116
4.6 Summary of the Chapter 117

5 AutumnLeaves – Reducing the Number of Open Windows 119
5.1 Introduction 119
5.1.1 Positioning *AutumnLeaves* 119
5.1.2 Introduction to *AutumnLeaves* 120
5.2 Problem Analysis: Window Plague in IDEs 122
5.3 AutumnLeaves 125
5.3.1 AutumnLeaves in a Nutshell 125
5.3.2 Variations, Modifications, Adaptations 128
5.4 Validation 130
5.4.1 Correctness 130
5.4.2 Practicality 136
5.4.3 Differences between IDEs 137
5.5 Summary of the Chapter 138

6 Discussion 139
6.1 Other IDE Enhancements Tackling Information Overload . 139
6.2 Conclusions 140
6.2.1 Problems Addressed 140
6.2.2 Remaining Problems 142

II Exploiting Dynamic Information in IDEs 145

7 Hermion – Extending Source Code Perspectives with Dynamic Information 149

7.1 Introduction . . . 149
7.1.1 Positioning *Hermion* . . . 149
7.1.2 Introduction to *Hermion* . . . 150
7.2 Dynamic Information in the IDE . . . 152
7.2.1 Scenario: Understanding a Complex System . . . 152
7.2.2 Hermion Overview . . . 157
7.3 Dynamic Information Gathering . . . 158
7.3.1 Partial Behavioral Reflection . . . 159
7.4 Validation . . . 160
7.4.1 Case Studies: Pier and OmniBrowser . . . 161
7.4.2 Efficiency . . . 163
7.4.3 Preliminary Empirical Evaluation . . . 163
7.5 Discussion . . . 165
7.6 Related Work . . . 166
7.6.1 Techniques Encompassing Dynamic Information . 166
7.6.2 Techniques Purely Based on Static Analysis . . . 167
7.7 Summary of the Chapter . . . 168

8 Senseo – High Level Augmentations of IDEs with Dynamic Information 171

8.1 Introduction . . . 171
8.1.1 Positioning *Senseo* . . . 171
8.1.2 Introduction to *Senseo* . . . 172
8.2 Motivation . . . 174
8.3 Integrating Dynamic Information in IDEs . . . 176
8.3.1 Architecture . . . 176
8.3.2 Dynamic Information . . . 177
8.3.3 Enhancements to the IDE . . . 178
8.4 Collecting Dynamic Information . . . 181

8.5 Validation . . . 184
8.5.1 Experimental Design . . . 184
8.5.2 Results and Discussion . . . 188
8.5.3 Threats to Validity . . . 192
8.6 Performance . . . 194
8.7 Related Work . . . 197
8.8 Summary of the Chapter . . . 199

9 CollView – Representing Dynamic Collaboration in IDEs 201
9.1 Introduction . . . 201
9.1.1 Positioning *CollView* . . . 201
9.1.2 Introduction to *CollView* . . . 202
9.2 Hidden Dynamic Collaboration . . . 204
9.3 Representing Dynamic Collaboration in the IDE . . . 207
9.3.1 Gathering Dynamic Information . . . 207
9.3.2 Explicit Dynamic Collaboration . . . 208
9.3.3 Enhancing Existing IDE Tools . . . 211
9.4 Validation . . . 212
9.4.1 Performance Benchmarks . . . 212
9.4.2 Developer Feedback . . . 214
9.5 Discussion . . . 215
9.6 Related Work . . . 218
9.7 Summary of the Chapter . . . 220

10 FeatureEnv – Visualizing Software Features in IDEs 223
10.1 Introduction . . . 223
10.1.1 Positioning *FeatureEnv* . . . 223
10.1.2 Introduction to *FeatureEnv* . . . 224
10.2 Problem of Feature Identification . . . 226
10.2.1 Explicitly Representing Features in the IDE . . . 226
10.3 *FeatureEnv*, a Feature-centric Environment . . . 228
10.3.1 Feature Affinity in a Nutshell . . . 228
10.3.2 Elements of *FeatureEnv* . . . 229

10.3.3 Maintaining Software with *FeatureEnv* 233
10.4 Validation 234
10.4.1 Introducing the Experiment 234
10.4.2 Hypotheses 235
10.4.3 Study design 235
10.4.4 Study Result 237
10.4.5 Threats to Validity 242
10.4.6 Study Conclusion 243
10.5 Discussion 243
10.6 Related Work 245
10.7 Summary of the Chapter 246

11 Discussion **249**
11.1 Problems Addressed in the Second Part 249
11.2 Problems Previously Addressed 251
11.3 Remaining Problems 252

III Conclusions **255**

12 Contributions **259**

13 Perspectives **265**

IV Appendices **269**

A Additional IDE Enhancements **271**
A.1 Visualizations 271
A.1.1 System Complexity View 272
A.1.2 Class Blueprint 273
A.1.3 UML Class Diagrams 275
A.2 Iconic Information 276

Bibliography **279**

List of Figures

1.1 Table of development activities and their problems developers face when working on them in IDEs. A cell with an 'X' means that the corresponding problem affects the corresponding activity, while a grayed out cell means that there is no special influence of this problem on this particular activity. 9

1.2 The different problems of IDEs our seven techniques tackle and to which development activities they thus contribute. A cell with an 'X' means that the corresponding problem affects the corresponding activity, but we do not provide a solution for this particular problem. A grayed out cell means that there is no influence of a problem on a particular activity. 18

2.1 Seesoft colors source code lines in a heat gradient to draw a developer's attention to important lines. 25

2.2 Seesoft tackles the information overload and the missing overview in IDEs, but only on a source code level. 26

2.3 Microprints appearing next to the method source code in the VisualWorks Smalltalk IDE. 27

2.4 Microprints mitigate the problem of information overload and missing overview in IDEs, but only on a method and single class level. 27

2.5 Fluid source code views inlining the method definition of the invoked method getNextTask() in Eclipse's source editor. 28

2.6 Fluid source code views aims at improving the information overload, the overview, and the access to distributed artifacts. 29

2.7 Hopscotch's expandable and collapsable source editor view showing several classes and their methods. 30
2.8 CodeSonar specifically addresses the problem of not having support for quality assessment in IDEs and also improves the overview of class relationships in software systems. 31
2.9 ROSE's suggestion view (lower right) integrated into Eclipse. 33
2.10 Hipikat's results view showing tasks similar to the currently performed task as specified in a task report. 34
2.11 ROSE, Hipikat, and other mining approaches tackle the problem of related but distributed artifacts whose collaboration is hidden in IDEs. Furthermore, they also mitigate the information overload problem as related artifacts can be easily navigated using the recommendation lists. 34
2.12 A concern representation in FEAT integrated in Eclipse. . . 36
2.13 FEAT tackles the problems of missing overview and information overload. If recorded navigation activities are accurate, FEAT can reveal hidden dependencies between distributed source artifacts relevant for specific features. . 37
2.14 NavTracks' related files view integrated in Eclipse. 37
2.15 NavTracks mitigates the information overload problem as related entities can be quickly navigated with the recommendation list, which also contains distributed artifacts whose collaboration is otherwise not explicit in the IDE. . . 38
2.16 The different views provided by Mylyn in Eclipse. 40
2.17 Mylyn highlights interesting artifacts to mitigate the information overload, identifies task-relevant entities, and, dependent on the quality of the development activities, reveals hidden collaboration between artifacts or identifies entities used in particular features. 41
2.18 Whyline improves the understanding of static source code, execution flow, and features. Hidden collaborations can be also spotted in some cases. 42
2.19 The method trace view of Compass visualizes the entire runtime control flow as a tree of nodes in a fisheye view. A node represents a method execution. The call stack below the method trace view focuses on a single slice of the trace. 43

2.20 Compass reveals hidden dependencies between distant source artifacts and improves understanding of static source code and execution flow in specific system executions. . . 44

2.21 An example of JQuery showing in Eclipse an exploration process tree starting with the results of a query. 46

2.22 JQuery reduces information overload in IDEs by explicitly representing concerns, thus relevant artifacts can be studied in a single perspective, which also improves the overview. Hidden collaboration between distributed artifacts is determined purely by static analysis. 46

2.23 Ferret's query results view integrated in Eclipse. 47

2.24 By providing a dedicated but often overloaded query view, Ferret improves to some degree information overload and overview in the IDE. For the currently selected artifact, related artifacts are revealed based on static and dynamic analysis. However, only method invocations are dynamically analyzed, thus support for the understanding of execution flow, static source code, and dynamic collaborations is limited. 48

2.25 Summary of the different IDE problems tackled by the presented related works. All problems are mitigated, but not any of them thoroughly. 49

3.1 *HeatMaps* highlight relevant artifacts to reduce the information overload and increase the overview in static source views. *HeatMaps* also provide limited support for the representation of context and helps developers to identify distributed artifacts that are conceptually related. As *HeatMaps* can also take into account dynamic information, they make execution paths more tangible by highlighting executed artifacts. 68

3.2 A color gradient from light blue to light red representing heat. 72

3.3 Two HeatMaps highlighting number of versions of source artifacts, top left, and recently browsed artifacts, bottom right. 73

3.4 Time-based color gradient. 74

3.5 Metrics-based color gradient. 75

4.1 *SmartGroups* primarily mitigate the problem of information overload, represent context in IDEs, and, to a limited degree, also make explicit hidden collaboration between distributed source artifacts. 92

4.2 *SmartGroups* view integrated on the left side of Pharo Smalltalk's system browser, the core of the Smalltalk IDE. 102

4.3 Procedure to determine the correctness of an identified task-relevant elements depending on its position. 111

5.1 *AutumnLeaves* primarily alleviates the problem of an overloaded workspace in IDEs, which, in turn, also gives developers a better overview of the system under investigation. 120

5.2 Eclipse supports tabbed browsing of the source space, but there is only space for a limited number of tabs; additional tabs are accessible in scroll list at the right. 123

5.3 Squeak Smalltalk provides a desktop on which full-fledged windows are opened, similar as in MacOS X. 123

6.1 The various lDE shortcomings addressed by the proposals presented in the first part of the dissertation (*HeatMaps*, *SmartGroups*, and *AutumnLeaves*) and the development activities to which these proposals contribute (HM = *HeatMaps*, SG = *SmartGroups*). 140

7.1 *Hermion* primarily addresses the problem of imprecise static source code and also of unclear execution flow in methods. Additionally, hidden collaboration between distributed artifacts is made explicit on a method and class level. 150

7.2 UML Class Diagram of the OmniBrowser kernel classes. . 153

7.3 Static search (1) vs. precise dynamic search (2) for implementors of *children* in *Hermion*. 155

7.4 List of methods invoked for message send *nodesFrom:forNode:* in *Hermion*. 155

7.5 List of types of instance variable *selection* extracted from dynamic information in *Hermion*. 156

7.6 Enriched method source code view including a reference view in *Hermion*. 157

7.7 The link invokes the metaobject upon occurrence of selected base-level operations. 161
7.8 Comparison of execution times for different levels of instrumentation for OmniBrowser and Pier. 162

8.1 *Senseo* contributes to a better system overview, makes visible dynamic collaboration between distant artifacts, improves the understanding of static source code and execution flow in and between source artifacts, and even offers limited support for quality assessment. 172
8.2 Setup to gather dynamic information. 177
8.3 Sample code and its corresponding CCT. 179
8.4 All six interactive views of *Senseo*. 179
8.5 Simplified excerpt of the CCTAspect 183
8.6 Box plots comparing time spent and correctness between control and experimental group. 190
8.7 *Senseo* overhead for the DaCapo benchmarks. 195
8.8 Size of transmitted data packets for "eclipse". Serialization/transmission rate: 1.25 packets per second. 196

9.1 *CollView* aims at explicitly representing and visualizing dynamic collaboration between related but statically distributed source artifacts. Moreover, *CollView* uncovers execution flow primarily on a method level but to some degree also on a class or package level. Eventually, *CollView* contributes to a better system overview by displaying collaboration on a package level. 202
9.2 UML diagram of Mondrian classes involved when displaying a graph. 204
9.3 Sequence diagram in Mondrian to display a graph. 206
9.4 Class Collaboration Chart generated by the IDE. 209
9.5 Package Collaboration Chart generated by the IDE. 210
9.6 Method Collaboration Chart generated by the IDE. 211
9.7 Integration of a class collaboration chart in the Squeak Smalltalk IDE. 211

10.1 *FeatureEnv* addresses the problem of the invisibility of features in IDEs by explicitly representing them and also contributes to make visible hidden collaboration between distributed source artifacts. 224

10.2 The relevant Pier class hierarchies for the *copy page* feature and its call graph. 227

10.3 The Elements of our *FeatureEnv*. 229

10.4 The Common Subexpression and Sequence Compression of the Feature Tree. 231

10.5 Comparing average time to correct the two defects. 238

10.6 Comparing average time between using *FeatureEnv* and OMNIBROWSER to discover and correct a defect. 238

10.7 Boxplots showing the distribution of the different subjects. 239

10.8 Comparing the average results for the effect of compact feature overview on program comprehension. 240

11.1 The various IDE shortcomings addressed by the proposals presented in the second part of the dissertation (*Hermion*, *Senseo*, *CollView*, and *FeatureEnv*) and the development activities to which these proposals contribute (H = *Hermion*, S = *Senseo*, CV = *CollView*, FE = *FeatureEnv*). 250

A.1 System complexity view of the AST package integrated in the Squeak OmniBrowser IDE. 273

A.2 Class blueprint of the RBBlockNode class. 274

A.3 UML class diagram of a part of the AST package. 275

A.4 Several icons appear when browsing class String, such as the abstract, overridden, overrides, or overrides and overridden icon. 278

List of Tables

1.1 Three indicators highlighting navigation issues caused by information overload in IDEs. 5

1.2 Information overload caused by too many open windows in the Eclipse and Smalltalk IDE. 6

3.1 Accuracy rates of different HeatMaps in the Monitoring Use Case. 79

3.2 Accuracy rates of different HeatMaps in the Historical Use Case. 80

3.3 Performance of different HeatMaps in specific tasks. . . . 83

4.1 Five indicators highlighting navigation issues occurring in the Squeak Smalltalk IDE. 95

4.2 The different parameters used in the algorithm to identify entities relevant for defect correction tasks and how they influence the order of the relevant entities. 103

4.3 The different parameters used in the algorithm to identify entities relevant for feature implementation and adaptation tasks and how they influence the order of the relevant entities. 104

4.4 The different parameters used in the algorithm to identify entities relevant for program comprehension tasks and how they influence the order of the relevant entities. . . . 105

4.5 The different parameters for considering dynamic information to refine the ranked list of relevant entities. 106

4.6 Results of the benchmark evaluation for defect correction tasks. 112

4.7 Results of the benchmark evaluation for feature implementation and adaptation tasks. 112
4.8 Results of the benchmark evaluation for program comprehension tasks. 113

5.1 Characteristic of the window plague in the Eclipse and Smalltalk IDE 124
5.2 Weight addition to source entities upon certain actions on the same or dependent entities. Propagation means adding weight to related entities, for instance from a method to its class or from a class to its superclass. 127
5.3 Weight addition to the a window upon certain actions on this window. 127
5.4 Correctness, false positives, false negatives and average number of windows improvements provided by *Autumn-Leaves* of three randomly selected sessions and averaged over all 25 sessions. 133

8.1 Average expertise in control and experimental group. . . . 185
8.2 The five software maintenance tasks. 187
8.3 Statistical evaluation of the experimental results. 189
8.4 Task individual performance concerning time required and correctness. 191
8.5 Percentage of subjects using specific dynamic information in particular tasks. 191
8.6 Mean ratings of the subjects for each feature of *Senseo*. . . . 192

9.1 Time to gather data and render a class collaboration chart for Mondrian. 214
9.2 Time to gather data and render a class collaboration chart for Pier. 214
9.3 User rating for asked statements during the experiment. . 215

10.1 Formulation of the null hypotheses. 235
10.2 Questionnaire. 240

12.1 How we validated each proposal and the outcome of these validations. 263

A.1 Icons available in Squeak Smalltalk for different source artifacts. 277

Chapter 1

Introduction

1.1 Problems of Traditional IDEs

Traditional integrated development environments (IDEs) such as Eclipse [ECLI 03] and NetBeans [NETB 10] for Java, Microsoft Visual Studio [MICR 10] for C#/C++, or VisualWorks [VISU 10], Pharo [BLAC 09] and Squeak [INGA 97] for Smalltalk usually provide a static perspective on an object-oriented software system. Such a perspective provides a means to read, modify, create, or delete static source artifacts such as packages, classes, or methods.

Object-oriented language features such as late-binding, inheritance, or polymorphism, however, usually lead to distributed and scattered code which is hard to understand by just focusing on static source artifacts and static relationships between these artifacts [DEME 03, DUNS 00, WILD 92, NIEL 89a, HAMO 05]. Often it is not possible to identify and locate conceptually related code in the static source space as many relationships are purely dynamic and thus only present at runtime [NIEL 89a, NIEL 89b, DUNS 00]. Due to the narrow focus of IDEs on static source perspectives, most of these dynamic relationships between source artifacts remain unclear, obscure or simply invisible to the developer while using the static perspectives of IDEs. In short, traditional IDEs lack dynamic information in their usually purely static source perspectives.

In today's IDEs developers are often overloaded with information. First, IDEs usually contain many complex perspectives and facilities such as search widgets, menus with hundreds of options, or a plethora of open windows or tabs [SING 05]. Second, the software systems maintained

in an IDE are typically large. A developer is overwhelmed by the vast amount of system artifacts (for instance, classes or methods, configuration or documentation files, log files, etc.) and has thus difficulties to gain an overview or an initial understanding of an unfamiliar system [SING 05, KERS 05, KERS 06]. Developers miss a "big picture" view of the system or a guide throughout the system, for instance by visually relating artifacts that conceptually belong together but are widely distributed in the static source space [DESM 06]. IDEs fail to give such a guidance how to overview and navigate a system. Hence, adding even more information to IDE perspectives to also show dynamic relationships between the static source elements is dangerous as such additional information might further obstruct system overview and navigation.

We realize that IDEs suffer from two main problems, namely *overloading developers with too much information* and yet at the same time *narrowly focusing on a software's static structure* and thus missing information about dynamic relationships between distributed source artifacts.

As software development is a complex process, we first identify several distinct activities in this process. The identified problems of IDEs do not affect all these development activities in the same way. Second, we carefully analyze the two main issues of IDEs to separate several sub-problems. Finally, we elaborate a taxonomy that reveals which IDE shortcoming affects which development activity such as feature implementation or artifact collaboration investigation. This taxonomy serves as a guideline to generate ideas how we can address the different shortcomings of IDEs tailored to the needs of different software development activities.

1.1.1 Development Activities

In the following, we introduce nine development activities proposed by the literature [PACI 04] as a comprehensive set of activities typically performed by developers during software maintenance, but also during initial development of applications. Most of the time, these activities are performed in IDEs. Sometimes developers additionally use other tools or environments, such as software analysis tools, but the IDE remains the primary development tool for practically all activities and developers. According to studies and surveys Eclipse is used by more than half of all Java developers [GOTH 05], while only a negligible percentage is using conventional text editors such as Emacs [CAME 96]. As most Smalltalk dialects come with a complete environment in which the IDE is included, it is clear that Smalltalk developers are using an IDE and not, for instance, a plain text editor.

We follow the development activities framework introduced by Pacione *et al.* [PACI 04] which proposes the subsequent nine core activities.

1. *Feature investigation.* In this activity, developers analyze how software features (or parts thereof) are implemented, for instance to reveal which artifacts collaborate to each other to realize a specific feature.

2. *Feature implementation and adaptation.* This activity is concerned with the implementation of new software functionality or the adaptation, extension, or improvement of existing software features.

3. *Artifact investigation.* When investigating artifacts, developers analyze the internal structure of packages, classes, or methods to, for instance, reveal the class-internal execution flow.

4. *Dependency investigation.* Studying the dependencies or relationships between different artifacts is called dependency investigation. This activity is for example performed when developers need to understand communication patterns between two classes to reveal the degree of coupling between them.

5. *Runtime interaction investigation.* In this activity, dynamic communication and interaction between different artifacts is analyzed, for example which messages are sent by instances of a class to instances of another class or how two packages interact at runtime, *e.g.* which classes of the two packages communicate with each other.

6. *Artifact usage investigation.* Developers investigate the usage of single artifacts to, for instance, reveal the clients of a specific artifact or how often this artifact is invoked in a specific software feature.

7. *Execution patterns investigation.* Developers investigate patterns in system's execution to better understand the running of the system, to reveal communication paths between artifacts or within a single artifact, to follow the control flow, or to assess performance issues, *e.g.* when a frequently occurring execution pattern is slow.

8. *Quality assessment.* Assessing a system's quality is necessary when the system is hard to maintain, evolve, or also when it has performance problems. When assessing software quality, developers usually perform one or several of the other activities as well.

9. *Domain concepts understanding.* To successfully implement, maintain and deploy a system, developers also need to understand its domain and how the domain concepts are represented and implemented in

the system. Thus this activity is concerned with studying, locating, or identifying domain concepts in the software system.

In the next section, we identify and categorize the main shortcomings of IDEs with respect to these software maintenance activities.

1.1.2 Problem Identification

While performing these development activities, developers are usually affected or even hampered by one or more shortcomings of traditional IDEs. Before being able to tackle the problem of the narrow focus of IDEs on static software structure by augmenting these perspectives with dynamic information, we have to reduce or better organize the (static) information presented to developers to not overload the source views even more. Hence, we first analyze the information overload problem developers suffer from in most IDEs [DE A 08] and derive consequent problems from this major issue of IDEs. Second, we deduce several subsequent issues caused by the narrow focus of IDEs on static source perspectives that neglect runtime information and collaboration between the static source artifacts.

We deliberately do not discuss in our analysis other problems of IDE that are not directly related to the two main IDE issues. For instance, IDEs usually do not support quality assessment of the developed application. Developers do not obtain automatic feedback from the IDE whether the current implementation of the system is sound or rather error-prone. The IDE could, for instance, analyze whether the code being currently written has any flaws such as being a duplication of other code or whether it violates commonly accepted principles such as design or best practice patterns. However, such issues are largely beyond the scope of this work.

Overloaded, unorganized views. Comprehending a software system is a prerequisite to improve, extend, or correct it. Being overloaded with too much information in an IDE, however, makes it difficult for a developer to understand the implementation and behavior of a system [SING 05, KERS 05]. One negative impact of information overload is a *loss of overview* of the system [KERS 05]: How is the system structured, what are the relations between the different parts, where and how is a particular feature implemented — these and other questions are difficult to answer in a huge software space.

The IDE as the primary tool to navigate software does not well support the process of dealing with a huge software space. It offers only few means such as a tree of hierarchically related source artifacts (for instance,

Indicator	Avg. of 20 sessions
Number of window switches	38.85
Number of entities revisited	35.10
Edit / navigation ratio	9.51%

Table 1.1: Three indicators highlighting navigation issues caused by information overload in IDEs.

packages containing classes that contain methods) to help developers to gain some degree of overview [SING 05]. However, there is no clear path drawn by the IDE through the huge forest of software entities, in particular there is usually limited or *no support for task-oriented programming* [KERS 05, SING 05]. The IDE does not reflect about the nature of the current task-at-hand, this means there is no guide whatsoever to advise developers how to complete the current task, for instance suggestions which particular entities they need to consider to correct a defect [SING 05].

Related to missing task-orientation is the *unavailability of context* in IDEs [DESM 06]. The current context is for instance the working set of entities the developer is focusing on, that is, the entities relevant for the current development task. This context also consists of distinct views on these entities, such as open debugger or inspector views and type hierarchies or source repository views. A working context is often just a subset of all currently open views or windows in an IDE, as developers usually do not regularly close windows unrelated to the current focus of development [RÖTH 09a]. Thus having an explicit and persistent representation of a working context would help developers to keep and to later on re-establish the focus on the task-at-hand [KERS 05].

We analyzed various development sessions [RÖTH 09b] to receive an impression of how seriously developers are hampered in practice by information overload in IDEs and their missing task-orientation or context representation. In the first study, we recorded navigation and modification activities from 20 distinct development sessions lasting for 30 minutes and performed in Squeak Smalltalk [INGA 97] by twelve different developers working on small or medium-sized applications with not more than 100 classes. As indicators for navigation difficulties caused by information overload, we consider the number of *window switches* (changing focus from one window to another), the number of *re-visits of source artifacts* purely for reading and understanding (without modification), and the *edit/navigation ratio* (ratio of edit actions compared to navigation actions). Table 1.1 shows for each indicator the results.

Metric	Eclipse	Squeak
Number of windows opened	35.84	25.74
Avg. number of open windows	16.68	14.29
Number of windows closed	10.35	12.96
Number of window switches	58.90	38.85

Table 1.2: Information overload caused by too many open windows in the Eclipse and Smalltalk IDE.

The values obtained in this study for the various indicators confirm the hypothesis that navigating the source space in an IDE is often difficult. Developers frequently have to switch between different windows; opening views on the same source artifacts several times is also a frequent incident, even in short development sessions lasting just half an hour. Moreover, developers usually spend quite some time until they are able to locate in a maintenance task the artifacts they actually want to modify to correct a defect, as the low edit/navigation ratio shows. All these figures demonstrate that in the particular sessions we studied, developers were probably overloaded with information as they were not able to appropriately identify and focus on the artifacts of interest.

In a second study, we focus on another reason for information overload. Not only the multitude of source artifacts and their scatteredness, but also the often huge number of windows opened during a development session leads to a loss of overview. To study this phenomena we recorded in Squeak and Eclipse the number of opened windows in total, the average number of open windows (measured in intervals of five minutes), the number of windows closed, and again the number of window switches. In this study we analyzed 22 development sessions lasting for half an hour each and performed by six different developers. Table 1.2 reports on the findings of this study.

These numbers highlight the fact that developers usually open many more windows than they close, thus the list of opened windows steadily grows. Psychological research reports that human beings are capable to cognitively handle seven plus or minus two distinct items at any given time [MILL 56]. As the average number of open windows is more than twice as high as seven, developers are not able to cognitively handle so many open windows, thus they are likely to ultimately lose the overview and their navigation efficiency is hampered. The high number of window switches is an indication for lost overview and confusion resulting from being overloaded with too many windows in the development workspace [RÖTH 09a].

Summary. Our analysis of overloaded, unorganized views on source code in IDEs shows that this issue has several aspects and causes such as

busy interfaces containing many options, buttons and icons, workspaces with many open windows or tabs, or, most notably, the large amount of source artifacts software systems typically encompass. Consequently, it is difficult to gain an *overview* of a system, which has been confirmed by the results of empirical studies of how developers use the IDE [RÖTH 09b, MURP 06]. A treatment for overloaded views could be the representation of context and task-related entities, since this allows developers to focus and a small subset of all system entities and thus considerably limits the amount of information they have to deal with. However, conventional IDEs *do not represent working context or tasks*.

Narrow focus on static source artifacts. As information about dynamic relationships and collaborations between source artifacts such as classes or methods is usually missing in most traditional IDEs, developers resort to debuggers, profilers or static analyses to reason about runtime behavior of software systems. However, the information revealed by debuggers and similar tools is volatile and focuses on specific system executions, basically the current call stack [POTH 07], thus failing to provide a comprehensive view on the system's dynamic collaboration patterns. The results of static analyses are permanently accessible and independent of specific executions, but are often imprecise and unspecific [ROUN 03, ROUN 04, DMIT 04a]. Statically searching for methods invoked at a specific call site in a huge system, for example, often yields many candidates, possibly also unrelated candidates never invoked at runtime from this call site.

Both debuggers and static analysis tools are thus not always able to identify conceptually related but statically *distant artifacts*. Debuggers do not make such relationships explicit and static analysis does often not find all relationships or relates entities that actually do not communicate with each other at runtime [ROUN 03]. It is hence difficult to reveal statically not visible and thus *hidden dynamic collaboration* between source artifacts in an IDE [KO 04, DE A 08, DESM 06], in particular in dynamically-typed languages where the static type of variables is unknown [RÖTH 08a]. At runtime, a class might communicate with classes not foreseeable in the static source code [KO 04]. Analogously, determining and navigating to all artifacts with which a given entity (such as a class) dynamically communicates, is not easy in most IDEs [KO 04].

As information about runtime types of variables or about the receiver types of message sends is important for the understanding of a method's source code, the *lack of dynamic information also leads to hard to understand code* [KO 04, WILD 92]. In Java, for instance, the use of interface or abstract types obfuscates the concrete runtime types of variables and makes it often hard to determine to which receiver types messages are sent at

runtime [WILD 92]. Another problem that arises when looking just at static source code is that the *execution flow is hidden* as the exact execution path is determined just at runtime [TAEN 89, WILD 92], for instance due to late binding [WILD 92]. If for instance a method executes branches of code based on the specific types of a variable, it is crucial to know the types to which this variable is bound during specific system executions to understand the execution flow in this method [WILD 92].

Dynamic information is in particular relevant for the understanding of higher-level artifacts such as software features [EISE 05]. On a feature-level, the lack of dynamic information is usually more serious than on a local source code level, because there is basically no static representation of features in source code; features only exist at runtime [EISE 05, GREE 06]. Hence a developer is often at a loss to identify the static artifacts (classes, methods, etc.) implementing a specific feature [WILD 95, EISE 03]. Also the dynamic interplay of the static artifacts implementing a feature is not visible in static source code, thus *features are just implicitly represented in code* [JERD 96, EISE 03, GREE 06]. However, mapping features to source artifacts and understanding how these artifacts interact, is crucial for feature comprehension [EISE 03, EISE 05, GREE 06].

Summary. The narrow focus of IDEs on static software structure in their perspectives has several negative consequences for developers: Conceptually related source artifacts are spread over the entire software space. Often their dynamic *collaboration is not explicitly represented* in IDEs, thus developers cannot identify these artifacts as being related. *Static source code is often hard to understand*, particularly in dynamic languages, when no runtime information about types or message sending is available. The intra- and inter-procedural *execution flow is not made visible* in IDEs and is hence difficult to reconstruct for developers. The understanding of software features is poorly supported by IDEs, since they *do not represent features as tangible entities* of software and do not make explicit the source elements implementing particular features.

These analyses of shortcomings and issues of conventional IDEs lead us to the formulation of the subsequent problem statement and the research question our work sets out to answer.

Problem statement. *Being overloaded with often unorganized or unfocused static information in software development environments (IDEs) on the one hand and an unavailability of dynamic information on the other hand, renders program comprehension and software maintenance difficult.*

Research question. *How can we tackle the information overload in IDEs and at the same time reasonably integrate dynamic information in the static source perspectives?*

1.1.3 Taxonomy of IDE Problems and Development Activities

As most IDE problems and shortcomings affect developers only during some activities, we elaborate in the following a taxonomy of development activities and problems of IDEs by mapping them to each other based on whether a specific problem affects the developer when performing a specific activity.

For each development activity, we list all identified problems of IDEs that are negatively impacting this specific activity. Figure 1.1 subsumes the mapping of IDE problems to development activities. In the following, we explain in detail these mappings depicted in Figure 1.1 by elaborating for each development activity how a specific problem of IDEs makes it difficult to perform this activity. Note that some development activities directly correlate to problems of IDEs, such as missing support for quality assessment.

Problem \ Activity		Feature investigation	Feature implement-ation	Artifact investigation	Dependency investigation	Runtime interaction investigation	Artifact usage investigation	Execution pattern investigation	Quality assessment	Domain concept analysis
Overloaded views	Information overload	X	X	X		X				X
	No overview	X	X		X				X	
	No context, task support		X							
I	Quality assessment support poor						X		X	X
Narrow focus on static source perspectives and views	Distributed artifacts	X		X	X	X		X	X	X
	Collaboration hidden	X		X	X	X		X	X	X
	Execution paths hidden			X	X	X	X	X		
	Imprecise static source code			X		X		X		
	Features hidden in code	X	X							X

Figure 1.1: Table of development activities and their problems developers face when working on them in IDEs. A cell with an 'X' means that the corresponding problem affects the corresponding activity, while a grayed out cell means that there is no special influence of this problem on this particular activity.

Feature investigation. When developers investigate features to better understand their implementation by, for instance, identifying the classes used in a feature, the information overload in IDEs which leads to loss of overview of the system seriously impedes the feature investigation process. Since features as purely dynamic software artifacts are not explicitly represented in an IDE, it is hard to locate in which source artifacts of a huge system they are implemented. Having an explicit and tangible representation of features in IDEs mitigates this problem. When looking at the implementation details of a feature, an explicit representation of hidden collaborations between artifacts that are often spread over the entire software space, could ease feature understanding.

Feature implementation. Developers implementing new features are also struck by the missing overview in IDEs as this makes it difficult to locate similar features as the one to be implemented, for instance to reuse parts of an existing implementation. Support for task-oriented programming, for instance by providing working sets of entities relevant for related features, would be useful during the implementation of new features. Of course, having an explicit feature representation in the IDE would be particularly useful for feature implementation.

Artifact investigation. When developers are analyzing specific source artifacts (*e.g.* classes or packages) and in particular the relationships between these artifacts, information overload clearly hampers artifact investigation. Typically, the IDE displays all classes in a package without emphasizing which classes would be important for the developer to focus on to gain an understanding for the package under investigation. Similarly, it would be useful to see in the IDE how a package collaborates dynamically with other, possibly distant artifacts. Gaining a low-level understanding of source artifacts is also difficult due to the lack of dynamic information. For developers it is a challenge to comprehend the source of a method in the absence of execution path and runtime type information.

Dependencies investigation. What holds for artifact investigation, is also true for dependency investigation. Here in particular the availability of dynamic information about collaboration between artifacts and execution paths in the IDE could be very useful. Additionally, having a better overview of the entire system would make the identification of dependent artifacts easier.

Runtime interaction investigation. Clearly, for the investigation of the interaction between source artifacts, it would be useful to have any kind

of dynamic information available in the IDE. The information overload in IDEs, however, seriously hampers the process of identifying runtime interaction patterns.

Artifact usage investigation. Assessing how and how often artifacts are used by other elements in a system is usually part of system quality assessment, thus better support for quality assessment is also beneficial for artifact usage investigation, as is information about collaboration patterns between artifacts and possibly also information about execution paths.

Execution pattern investigation. To better support this development activity in the IDE, it is very useful to have dynamic information revealing the execution paths in and between source artifacts. Such information should be encompassed with information about the dynamic collaboration between distant artifacts.

Quality assessment. Besides not having direct and elaborated support for quality assessment in the IDE, this activity also suffers from lack of overview, which makes it difficult to identify problematic patterns or parts of the code that seriously affect the system quality. Making visible hidden collaborations could reveal communication patterns that break encapsulation or that impose other quality problems such as an unnecessarily tight coupling between two possibly distant artifacts.

Domain concept analysis. Higher level information such as an explicit representation of features or support for a better overview of the system directly in the IDE is useful to understand the domain concepts of a system. The information overload problem developers suffer from in most IDEs, however, makes domain concept analysis difficult to perform. The lack of information about dynamic collaboration and the missing link between artifacts that are conceptually related but statically distributed over multiple packages also hampers the identification of certain domain concepts in the software space. Of less interest but still useful is a representation of context in the IDE, for instance developers could build working sets representing the different domain concepts employed in a system.

Throughout the chapters of this work, each presenting a different approach to address different problems of IDEs, we always come back to the taxonomy of IDE problems and development activities. This serves as a roadmap for the entire dissertation.

1.2 Proposal: Tackling Overloaded Views and Integrating Dynamic Information in IDEs

We first introduce the thesis of this work and subsequently elaborate on how we substantiate this thesis by presenting proposals aiming at addressing the aforementioned problems.

We formulate our thesis as follows:

Thesis

To effectively use dynamic information for software maintenance tasks, we first need to mitigate the information overload in the static views of IDEs on source code, and subsequently augment these existing and familiar views on the software structure with dynamic information.

In the following two sections, we outline the approaches we implemented to support our thesis. Initially, we present three approaches that tackle the information overload problem. Based on the improvement achieved by the first three approaches, we are then in a position to propose four distinct contributions that address and tackle the issue of missing dynamic information in IDEs. Some of the approaches we describe actually address both of the above problems.

1.2.1 Mitigating Information Overload in IDEs

All three approaches alleviating the information overload problem of IDEs have been implemented in the Smalltalk IDE (Squeak and Pharo) and some are also available for the Eclipse Java IDE.

***HeatMaps* – A Navigational Aid.** *HeatMaps* mitigate the negative impact of information overload by highlighting relevant artifacts and thus easing system navigation. For this, *HeatMaps* color relevant artifacts (packages, classes, methods, etc.) in a software system with a heat color from blue to red. The more red an artifact is colored, the more important this artifact is considered to be for the developer. The importance value is determined with various combinable metrics such as how often or recent an artifact has been modified or navigated, how extensively it has been modified, or by which developer it has been committed how often. Importance is determined differently depending on the nature of the task developers are performing.

HeatMaps primarily mitigate the information overload in IDEs by helping developers to focus in a large software space on the source artifacts likely to be relevant. Furthermore, *HeatMaps* provide a quick overview of the system as the coloring by relevancy allows developers to efficiently identify interesting elements in a large software system. *HeatMaps* also provide a form of context; if for instance the 'recently modified' metric is used to color artifacts, *HeatMaps* highlight the context of recent modification, which is interesting when having to fix a recently introduced defect. Eventually, *HeatMaps* also contribute to the problem of distant but related artifacts. Such artifacts can be quickly identified as they often change in tandem, thus *HeatMaps* color them red.

An implementation of *HeatMaps* is available for the Squeak and Pharo Smalltalk IDE.

***SmartGroups* – Representing Context in IDEs.** *SmartGroups* mitigate the information overload by representing context in the IDE, thus allowing developers to focus on a small part of the possibly huge software space. There are three different kinds of smart groups: (i) manual smart groups whose elements are added manually by the developer, (ii) smart groups making search results permanently available, and (iii) automatic smart groups whose elements are automatically categorized. *SmartGroups* automatically categorize entities relevant for a particular type of task (either defect correction, feature implementation, or general program comprehension task) by exploiting recorded previous development activities, evolutionary information, and dynamic information. *SmartGroups* use algorithms similar to those of *HeatMaps* to identify task-relevant entities.

SmartGroups mitigate the information overload in IDEs as developers can focus on the working sets and hence do not have to navigate the entire, potentially large software space. Instead they just work in the *SmartGroups* view which contains a fraction of the complete software space. *SmartGroups* can represent dynamically related but statically distant entities, thus this approach also addresses the problem of hidden collaborations between distant and distributed artifacts. *SmartGroups* can even take into account dynamic information to, for example, automatically build a smart group for each artifact used in a specific system execution.

An implementation of *SmartGroups* is available for the Squeak and Pharo Smalltalk IDE.

***AutumnLeaves* – Reducing the Number of Open Windows.** The third approach, *AutumnLeaves*, provides "housekeeping services" for the IDE workspace by automatically identifying and removing unused open windows or tabs in a development environment. Typically, developers open many, possibly unrelated windows or tabs in an IDE even during short development sessions. These open views then clutter and overload the workspace, causing the developer to lose the overview. *AutumnLeaves* continuously analyzes all open windows and computes based on their content the relationships between them. If a window seems unrelated to most or all other open windows, that is, to the current focus of the developer, *AutumnLeaves* suggests to close this particular window. The degree of "unrelatedness" to the prevailing development focus is continually displayed for each window.

As *AutumnLeaves* reduces the number of open views (windows or tabs) in an IDE, this approach improves the overview in the IDE and thus also mitigates the information overload as developers have to deal with fewer open windows or tabs.

AutumnLeaves is available for the Squeak and Pharo Smalltalk IDE and also for the Eclipse Java IDE.

1.2.2 Enhancing IDEs with Dynamic Information

We contribute four distinct techniques to integrate dynamic information in different ways into development environments. The four approaches presented in the following address the previously identified problems caused by lack of dynamic information (cf. Section 1.1.2).

***Hermion* – Extending Source Code Perspectives with Dynamic Information.** *Hermion* augments the understanding of static source code by embedding dynamic information in the source code views. *Hermion* focuses on dynamically typed languages such as Smalltalk and particularly exploits dynamic information like runtime types of variables, receiver and argument types of message sends, or callers of a particular method. This information is integrated in the standard source code perspectives of the IDE by using tooltips that appear when holding the cursor over a piece of code (*e.g.* a variable) or by placing small, clickable icons next to the code statements in source code to trigger popup windows showing dynamic information. Additionally, developers can use the presented dynamic information for navigation, for instance to navigate to the methods invoked at runtime at a particular call site in a method's source code. Furthermore, next to each method and class, *Hermion* displays a list of all

types that have been used or referenced in this entity during a system's recorded execution(s).

Thus *Hermion* mostly addresses the problem of difficult to understand static source code and execution paths in this code; particularly in dynamic languages where variables have no statically defined type, the availability of type information in the static source views is very useful. Furthermore, *Hermion* makes collaboration between artifacts more tangible and accordingly helps developers locating distributed artifacts.

Hermion is available for the Squeak Smalltalk IDE.

***Senseo* – Augmenting Static Source Perspectives of IDEs with Dynamic Information.** *Senseo* aims at increasing the understanding of dynamic collaboration between distributed static source artifacts. For this purpose, *Senseo* offers a collaboration view for each artifact to reveal all system artifacts that use or are used by this artifact ("callers" and "callees", respectively). *Senseo* also enriches the source code views with information about types of variables and receiver or argument types of message sends. Additionally, it reports on how often artifacts are used in specific system executions, how many objects have been created in a method or class, or how many bytecode instructions have been executed by particular entities (methods, classes, packages).

Senseo mainly tackles the problem of hidden collaboration between distant source artifacts by making such collaboration explicit. Moreover, *Senseo* helps developers to understand executions paths and static source code. Eventually, *Senseo* also addresses the information overload issue by providing an overview of the entire system with respect to specific dynamic metrics. The collaboration view, for instance, improves the overview of the system by revealing collaborations between entire packages.

Senseo is available for the Eclipse Java IDE.

***CollView* – Representing Dynamic Collaboration in IDEs.** *CollView* explicitly represents dynamic collaboration between source artifacts in the IDE by providing visualizations of collaboration patterns. There are three different kinds of collaborations visualized by *CollView*, namely collaboration between packages, classes, and methods. For each of these three types of artifacts, we display next to a particular artifact the corresponding collaboration view. For packages and classes, the visualization focuses on the artifact (class or package) of interest and then draws edges to all artifacts with which the selected artifact collaborates (callers and callees). The more collaboration between two artifacts occurs at run-

time (measured by number of method invocations between them in the recorded system execution), the closer they are displayed and the thicker the line between them. For method collaboration, we opted for a similar layout as used in the UML sequence diagram. *CollView* particularly addresses a limitation of *Senseo*, namely that *Senseo*'s collaboration view is rather difficult to use for navigation and provides just a limited overview of the collaboration patterns.

CollView primarily aims at making collaboration between remote and distributed artifacts visible. Additionally, *CollView* also supports developers in better understanding execution paths and static source code, in particular thanks to the visualization of collaboration patterns on the method level. Similarly to the collaboration view in *Senseo*, *CollView* also contributes to a better overview of the system, particularly with the package collaboration view.

CollView is available for the Squeak and Pharo Smalltalk IDE.

***FeatureEnv* – Visualizing Software Features in IDEs.** *FeatureEnv* explicitly represents entire software features in the IDE. For this, *FeatureEnv* visualizes all artifacts used in one or several features. To reveal these used artifacts, the developer runs the system and executes the features of interest while the enhanced IDE analyzes the system execution. In *FeatureEnv*, we visually compare several features to each other, because such a comparison is often useful to detect anomalies (*e.g.* bugs) in a specific feature. This comparison also improves feature comprehension by relating one feature to other, similar features. A single feature can be visualized as an interactive method call tree to study the implementation of this particular feature. Furthermore, *FeatureEnv* provides an adapted version of the code browser highlighting all entities used in the feature to visually separate them from other system entities.

FeatureEnv mainly addresses the problem of not having an explicit feature representation in IDEs, but also aids developers in locating hidden collaboration between distant and distributed artifacts by connecting such artifacts in the feature visualizations.

FeatureEnv is available for the Squeak Smalltalk IDE.

1.2.3 Summary

To summarize this section, we enhance Figure 1.1 by also indicating which of the previously mentioned seven approaches address which IDE problem. By doing so, we also indicate which approach contributes to which development activity. For instance, *Senseo* mostly contributes to

activities such as feature investigation or implementation, dependency investigation, runtime interaction investigation, or execution pattern investigation. Figure 1.2 shows the mapping of development activities to IDE problems and indicates which of our approaches address a particular problem. An 'X' in this mapping indicates that we do not contribute a cure for a specific problem.

Problem \ Activity		Feature investigation	Feature implement-ation	Artifact investigation	Dependency investigation	Runtime interaction investigation	Artifact usage investigation	Execution pattern investigation	Quality assessment	Domain concept analysis
Overloaded views	Information overload	*HeatMaps SmartGroups AutumnLeaves*	*HeatMaps SmartGroups AutumnLeaves*	*HeatMaps SmartGroups AutumnLeaves*		*HeatMaps SmartGroups AutumnLeaves*				*HeatMaps SmartGroups AutumnLeaves*
	No overview	*HeatMaps AutumnLeaves Senseo CollView*	*HeatMaps AutumnLeaves Senseo CollView*		*HeatMaps AutumnLeaves Senseo CollView*				*HeatMaps AutumnLeaves Senseo CollView*	
	No context, task support		*SmartGroups HeatMaps*							
I	Quality assessment support poor						X		X	X
Narrow focus on static source perspectives and views	Distributed artifacts	*HeatMaps SmartGroups Hermion Senseo CollView FeatureEnv*		*HeatMaps SmartGroups Hermion Senseo CollView FeatureEnv*	*HeatMaps SmartGroups Hermion Senseo CollView FeatureEnv*	*HeatMaps SmartGroups Hermion Senseo CollView FeatureEnv*		*HeatMaps SmartGroups Hermion Senseo CollView FeatureEnv*	*HeatMaps SmartGroups Hermion Senseo CollView FeatureEnv*	*HeatMaps SmartGroups Hermion Senseo CollView FeatureEnv*
	Collaboration hidden	*HeatMaps SmartGroups Hermion Senseo CollView FeatureEnv*		*HeatMaps SmartGroups Hermion Senseo CollView FeatureEnv*	*HeatMaps SmartGroups Hermion Senseo CollView FeatureEnv*	*HeatMaps SmartGroups Hermion Senseo CollView FeatureEnv*		*HeatMaps SmartGroups Hermion Senseo CollView FeatureEnv*	*HeatMaps SmartGroups Hermion Senseo CollView FeatureEnv*	*HeatMaps SmartGroups Hermion Senseo CollView FeatureEnv*
	Execution paths hidden			*HeatMaps Hermion Senseo CollView*	*HeatMaps Hermion Senseo CollView*	*HeatMaps Hermion Senseo CollView*	*HeatMaps Hermion Senseo CollView*	*HeatMaps Hermion Senseo CollView*		
	Imprecise static source code			*Hermion Senseo*		*Hermion Senseo*		*Hermion Senseo*		
	Features hidden in code	*FeatureEnv*	*FeatureEnv*							*FeatureEnv*

Figure 1.2: The different problems of IDEs our seven techniques tackle and to which development activities they thus contribute. A cell with an 'X' means that the corresponding problem affects the corresponding activity, but we do not provide a solution for this particular problem. A grayed out cell means that there is no influence of a problem on a particular activity.

1.3 Contributions

This work aims at mitigating the two main problems of traditional IDEs, namely information overload and the unavailability of dynamic information, and their subsequent sub-problems. The following contributions of this work pursue this goal:

1. *HeatMaps* [RÖTH 09c] highlight in the Squeak or Pharo Smalltalk IDE entities relevant for specific software maintenance tasks. Besides the approach itself, we also contribute a validation of this approach by applying it to recorded data sets of navigation and modification activities performed by developers in several software systems. We refer to this kind of validation as *benchmark validation*.

2. *SmartGroups* [RÖTH 09b] automatically represent in the Squeak or Pharo Smalltalk IDE working context, that is, groups of task-relevant source entities. We validated *SmartGroups* with a benchmark validation similar to the one performed for *HeatMaps*.

3. *AutumnLeaves* [RÖTH 09a] provides "housekeeping" services in the Squeak or Pharo Smalltalk IDE and in the Eclipse Java IDE by automatically identifying and closing unused views (windows or tabs) in the IDE workspace. We contribute a benchmark validation accompanied with practical user feedback to evaluate *AutumnLeaves*.

4. *Hermion* [RÖTH 08a, RÖTH 08b] augments the understanding of static source code by embedding directly in the static source perspectives of the Squeak and Pharo Smalltalk IDE various kinds of dynamic information, such as receiver and argument types of message sends or runtime types of variables. We validated *Hermion* with a user study to gather qualitative feedback about its practicability and usefulness.

5. *Senseo* [RÖTH 09d, RÖTH 09e] increases the understanding of dynamic relationships between distributed static source artifacts in the Eclipse Java IDE by integrating a collaboration view linking these conceptually related artifacts together. Besides the approach, we contribute a comprehensive validation of *Senseo* by means of a controlled empirical experiment with 30 industrial software developers that solved typical software maintenance tasks. This experiment reveals a statistically significant improvement of correctness and reduction of time spent solving the tasks when using *Senseo* compared to just relying on the traditional Eclipse IDE.

6. *CollView* [RÖTH 08c] visualizes in the Squeak Smalltalk IDE dynamic relationships between packages, classes, and methods to make hidden collaboration and execution patterns between distributed artifacts visible. We informally validated *CollView* by conducting interviews with developers.

7. *FeatureEnv* [RÖTH 07a, RÖTH 07b] visually presents software features in the Squeak Smalltalk IDE to improve feature comprehension and maintenance. We validated *FeatureEnv* by conducting a controlled empirical experiment with twelve developers that corrected two different defects in a large, unfamiliar software system, one with the *FeatureEnv* and one with the traditional Smalltalk IDE. We were able to measure a statistically significant improvement of defect correction time when using the *FeatureEnv*.

8. Additionally, we contribute several minor extensions and enhancements to IDEs such as the integration of visualizations [RÖTH 07c] (for instance, class blueprints enhanced with dynamic metrics). Furthermore, we contribute a comprehensive analysis and taxonomy of the major shortcomings, problems, and issues of traditional IDEs that hinder developers when working on software maintenance tasks. We map these shortcomings or problems to typical development activities to understand which problems of IDEs hamper developers during which kind of development activities.

1.4 Structure of the Dissertation

This dissertation is subdivided into three parts: The first part discusses our approaches addressing information overload, missing overview, and missing representation of context in IDEs. The second part covers our proposals integrating and exploiting dynamic information in IDEs. Finally, we conclude our work in the third part. Directly after this chapter in this introductory part of the dissertation, we elaborate in Chapter 2 the state of the art concerning research on development environments, program analysis, and software visualization.

Part I: Mitigating Information Overload in IDEs. The first part starts in Chapter 3 by introducing the *HeatMaps* approach which highlights source artifacts in IDEs according to their degree of interest for the current task the developer is performing.

Chapter 4 presents *SmartGroups* which support the manual and automatic categorization of source artifacts.

Chapter 5 presents *AutumnLeaves*, which automatically identifies in the IDE open windows or tabs that are not anymore relevant for the current development task and thus should be closed.

Finally, we conclude this first part in Chapter 6 by discussing and comparing the three presented proposals and by briefly introducing other techniques we developed to mitigate the information overload in IDEs.

Part II: Exploiting Dynamic Information in IDEs. Chapter 7 begins the second part with introducing *Hermion*, an enhancement to the Smalltalk IDE integrating dynamic information such as variable or argument types in source code views.

Chapter 8 discusses *Senseo*, an approach to augment the IDE with dynamic collaboration information by, for instance, displaying all dynamic callers or callees of a given source artifact.

Chapter 9 presents *CollView*, an approach to explicitly represent and exploit dynamic collaboration between source artifacts by visualizing the communication patterns between packages, classes, or methods.

Chapter 10 introduces *FeatureEnv*, an extension to the IDE which allows developers to visualize features to compare them to each other or to study their internal implementation.

Chapter 11 concludes the second part by critically comparing and evaluating each of the four presented approaches.

Part III: Conclusions. In Chapter 12 we conclude the dissertation with regard to whether we were able to positively answer the research question of this work. As we do not claim to have solved all issues of development environments, we also thoroughly analyze in Chapter 13 the perspectives and challenges still remaining to further improve the support for software maintenance activities in development environments.

Chapter 2

State of the Art

In this chapter we present the state of the art in research about development environments, software analysis (in particular dynamic analysis), and software visualization. We hereby focus on works related to the various approaches introduced in Chapter 1, and to be thoroughly discussed in subsequent chapters. The main focus is on related work in the area of development environments; we discuss proposals and enhancements to IDEs applying program analysis, source history analysis, and development activity analysis. We also report on debugging, profiling, and querying tools integrated in IDEs.

2.1 Development Environments

We structure our discussion of related work around the techniques used to better support program comprehension and software maintenance in IDEs. We identified three principal analysis techniques exploited by different approaches to provide developers with helpful additional information in IDEs: (i) program analysis, that is, analyzing structure and behavior of software systems to extract information useful for program understanding, (ii) source history analysis which explores the history of the software system (for instance, by mining software repositories) to find interesting patterns such as source artifacts changed in tandem, and (iii) developer activity analysis where navigation or modification actions performed by developers in the IDE are recorded and analyzed, for instance to discover entities likely to be related as they have frequently been navigated together. Some approaches use a combination of these

three analysis techniques. We discuss such approaches in the category we think they best fit. Other approaches only loosely employ an analysis technique but rather improve the presentation of the static source code in the IDE.

Apart from these approaches extending the IDE based on different analysis concepts and data sources, we also discuss techniques such as debuggers, profilers, and querying and exploration tools that often use a combination of different data sources and corresponding analyses to augment the IDE with information helping developers in program comprehension.

2.1.1 Program Analysis and Sophisticated Information Presentation

Traditional IDEs provide source perspectives showing purely static information about a software system. The hierarchical relationships of source artifacts (for instance, based on packages containing classes containing methods) are represented with tree views such as the package tree in Eclipse or the column-based system browser in Smalltalk in which each column represents one level of the static hierarchy of a software system (usually the package, class, method protocol, and method level). Such column-based browsers often just support the viewing and editing of one single element (*e.g.* a method) at a time. Editing a source element puts the entire browser in a mode which cannot be left without either saving the changes done in the edit mode, or discarding these changes. This problem is addressed by extensions supporting tabbed browsing where several views (*i.e.* tabs) on source elements can be opened in the same browser instance. While one or several tabs are in edit mode, developers can still navigate the source space using other tabs. Another approach to solve this "edit mode" problem is contributed by Hopscotch [BYKO 08] (discussed below in detail) which offers a modeless environment for manipulating source code.

Many IDEs also provide additional tools other than a tree- or column-based package browser, for instance a type or call hierarchy view, a call graph browser (*e.g.* the Source Navigator IDE[1]), or advanced search facilities. All these tools, browsers, perspectives, or views available in IDEs have in common that they purely exploit the static structure of software systems.

However, there are several techniques that extend or complement these simple views on static software structure. Furthermore, a few tech-

[1] http://sources.redhat.com/sourcenav

niques also exploit dynamic information to extend the source perspectives. We present some of these approaches in the following.

Seesoft [EICK 92] is a software visualization system that eases software analysis by mapping each line of code to a colored row. The color indicates an interest metric using a heat map approach: red lines are for instance most recently changed lines and blue lines least recently changed (see Figure 2.1). Seesoft explores different data sources, namely static or dynamic analysis (mainly profiling), but also version control information such as age or author(s) of source artifacts.

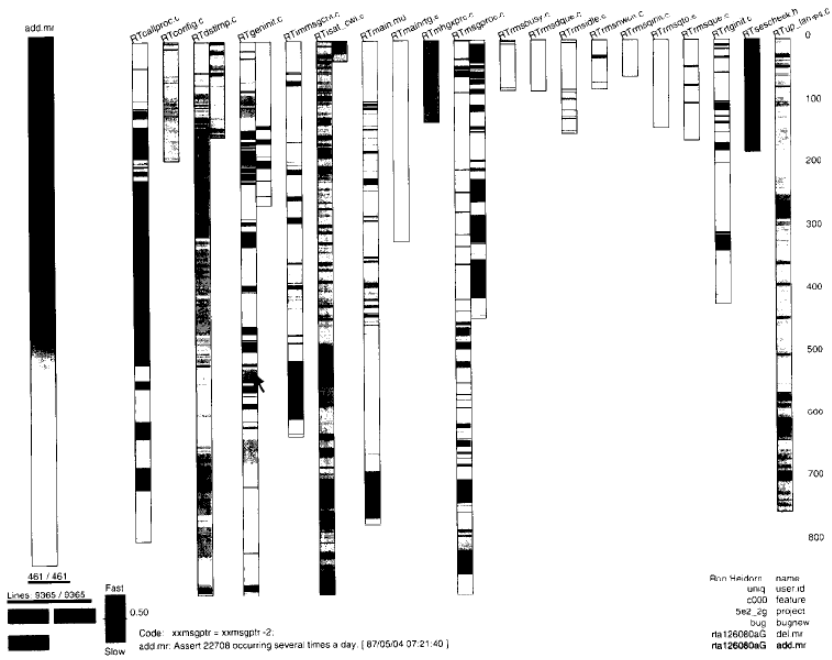

Figure 2.1: Seesoft colors source code lines in a heat gradient to draw a developer's attention to important lines.

Seesoft tackles the information overload problem in IDEs by visually highlighting source code lines that are likely to be of interest to the developer. Instead of having to read all source lines of a method, the developer quickly spots interesting lines due to the coloring applied by Seesoft. Hence the developer has to deal with less information, as the important information can be more quickly identified. Similarly, Seesoft also tries to improve the overview of a system. However, as this approach focuses

on single lines of code, it might help developers to more quickly gain an overview of a single method, but as systems encompass many thousand methods that are usually rather small, Seesoft is not able to provide an overview of a typical object-oriented system. Seesoft has its origin in procedural programming where functions typically consist of many more lines of code than methods in object-oriented applications. Accordingly, the reduction of information overload contributed by Seesoft is limited to single methods. Seesoft does not reduce the amount of source artifacts developers have to deal with while maintaining object-oriented software systems. Figure 2.2 summarizes the IDE problems addressed by Seesoft.

Problem tackled / Approach	Information overload	No overview	No context, task support	Quality assessment support poor	Dist-ributed artifacts	Colla-boration hidden	Execution paths hidden	Imprecise static source code	Features hidden in code
Seesoft	Only on source code level	Only on source code level							

Figure 2.2: Seesoft tackles the information overload and the missing overview in IDEs, but only on a source code level.

Microprints [DUCA 05a] are pixel-based character-to-pixel representations of methods enriched with semantic information mapped to specific colors (for instance, local variables are colored purple, return statements red) . Contrary to Seesoft [EICK 92], Microprints provide object-oriented specific information and visualize method semantics such as state access, control flow, or invocation relationships [DUCA 05a]. Microprints appear next to the method source code in the VisualWorks Smalltalk IDE as shown in Figure 2.3.

Microprints aim at allowing developers to quickly spot patterns, such as whether a method relies on superclass behavior, by mapping distinct colors to different types of message sends, *e.g.* messages sent to `super` are colored in orange. Variables or control statements are also mapped to distinct colors. Thus Microprints improve the overview of methods and entire classes as Microprints of all methods can be displayed in a row to better understand the interaction of a class with other classes in the hierarchy. Additionally, Microprints also tackle the information overload by saving developers from the need to read the source code line by line to identify patterns in methods, for instance to compare the implementation of two methods. As Microprints just statically exploit source code and method invocations relationships but not behavioral information, they do not improve the understanding of execution flow or static source code,

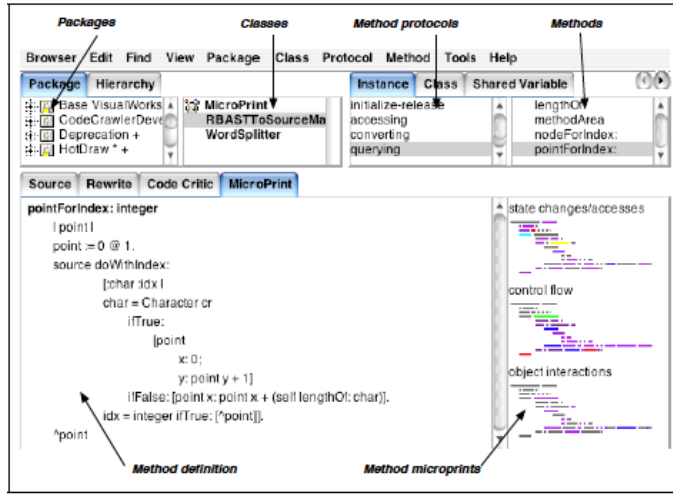

Figure 2.3: Microprints appearing next to the method source code in the VisualWorks Smalltalk IDE.

they just make the static information easier and faster to grasp. Figure 2.4 summarizes the IDE problems mitigated by Microprints.

Problem tackled / Approach	Information overload	No overview	No context, task support	Quality assessment support poor	Dist-ributed artifacts	Colla-boration hidden	Execution paths hidden	Imprecise static source code	Features hidden in code
Microprints	Only on method and single class level	Only on method and single class level							

Figure 2.4: Microprints mitigate the problem of information overload and missing overview in IDEs, but only on a method and single class level.

Fluid source code views [DESM 06] can embed related code (for instance an invoked method) directly in the current source code editor of Eclipse (see Figure 2.5). Developers can choose to view related code in the same source view by clicking on an expanded icon next to method invocation declarations in a method. The embedded remote code is not editable and appears colored to indicate that it is supplemental to the primary document [DESM 06]. It is possible to also extend method invocations in the embedded method definitions to view entire method invocation chains in the source editor [DESM 06]. Fluid source views recognize the separated but linked nature of source artifacts and support developers

in studying invoked code without having to navigate and thus change context. However, fluid source code views statically link separated source artifacts together and may thus identify wrong or unrelated candidate methods at polymorphic call sites. Not being able to modify the embedded source code we consider as a serious drawback. The ability to directly edit a related method at the place where it is invoked would be a very handy extension of the currently read-only fluid source code views.

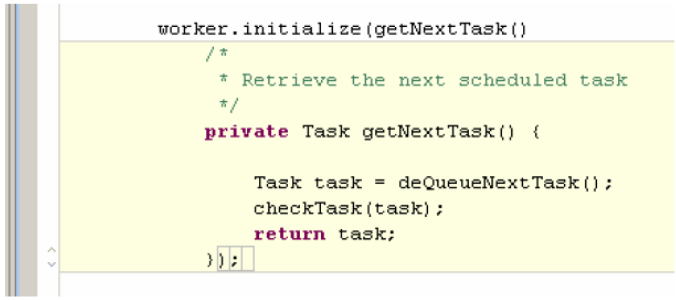

Figure 2.5: Fluid source code views inlining the method definition of the invoked method getNextTask() in Eclipse's source editor.

Fluid source code views aim at addressing the problem of having to maintain distributed artifacts, but these views only exploit static information, thus they cannot precisely link artifacts to each other in all cases. Higher level elements such as classes or packages are not considered, just methods are linked. The communication between linked elements is not hidden or implicit, instead it is clearly stated in source code. However, these views make it easier for the developer to look at the source code of remote methods, such as methods invoked by the currently selected method. Thus, fluid source code views also contribute to a better overview of the system and reduce the number of context switches and hence also of windows opened by developers, therefore reducing the information overload. Figure 2.6 subsumes the IDE problems mitigated by fluid source code views.

Hopscotch [BYKO 08] is the development environment of Newspeak, a programming language descended from Smalltalk and Self. In Hopscotch's source view, classes are initially shown as headers that can be expanded to see all defined methods or instance variables. Methods appear collapsed at first, but can be expanded to view and edit their code (see Figure 2.7). The Hopscotch editor provides a better overview than a flat file editor as developers can have an overview of all (collapsed)

Problem tackled / Approach	Information overload	No overview	No context, task support	Quality assessment support poor	Dist-ributed artifacts	Colla-boration hidden	Execution paths hidden	Imprecise static source code	Features hidden in code
Fluid source code views	Fewer windows	Limited support			Statically linked methods				

Figure 2.6: Fluid source code views aims at improving the information overload, the overview, and the access to distributed artifacts.

methods of a class and just dive in the source of those that are interesting for the current task. As several classes are shown in the same editor view, switching from a method of one class to a method of another is usually fast. Hyperlinks to quickly see all statically computed callers or callees of a method are also available. Inspired by web browsers, Hopscotch also provides back and forward buttons, for instance to come back to previously selected source artifacts when having navigated away to other classes or to callers of a particular method.

Hopscotch particularly tackles the problem of having too many modes in an IDE, for instance an edit mode that cannot be left without saving the changes. In Hopscotch, navigating away from an element with unsaved changes and coming back to these changes is always possible. However, we are skeptical whether the Hopscotch views improve the overview of a system. Being able to navigate several classes and their methods in the same editor usually leads to a huge editor view which must be scrolled all the time. Of course, open method definitions can be easily collapsed to make the view more compact, but the constant need to expand and collapse the view is cumbersome. While hyper-linking statically related source artifacts is certainly useful to ease navigation, it neither provides more overview or focus nor does it reduce the information overload. The Hopscotch IDE is an interesting improvement of the Smalltalk column-based IDE, but besides mechanisms to expand or collapse source elements, it does not deliver new ideas for tree- and file-based IDEs such as Eclipse, which does not suffer from the mode-related problems of the Smalltalk IDE that Hopscotch addresses. Ultimately, Hopscotch does not tackle any of the IDE problems identified in Section 1.1.2.

CodeSonar [RECH 07] automatically detects and visualizes quality defects in object-oriented software systems and is implemented as an Eclipse plugin. CodeSonar reasons about different types of static relations between source artifacts such as “extends”, “is_type_of”, or “return_type” to discover quality defects such as high coupling between classes. CodeS-

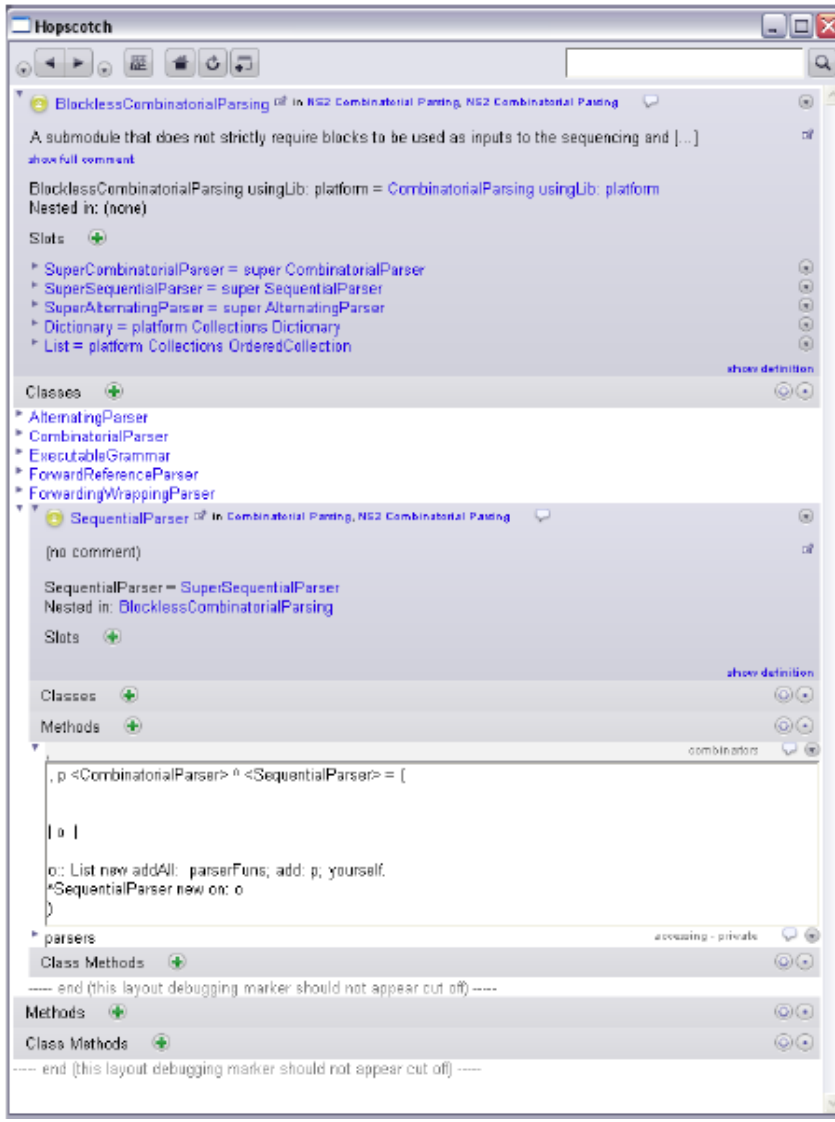

Figure 2.7: Hopscotch's expandable and collapsable source editor view showing several classes and their methods.

onar presents quality defects in navigable graphs visualizing classes as nodes and the aforementioned relations as edges.

Problem tackled / Approach	Information overload	No overview	No context, task support	Quality assessment support poor	Dist-ributed artifacts	Colla-boration hidden	Execution paths hidden	Imprecise static source code	Features hidden in code
CodeSonar		Only for classes and their static relations		For class level quality defects					

Figure 2.8: CodeSonar specifically addresses the problem of not having support for quality assessment in IDEs and also improves the overview of class relationships in software systems.

CodeSonar mainly addresses the problem of missing support for quality assessment in IDEs and provides with its graphs limited support for gaining an overview of a system (cf. Figure 2.8).

Summary. We note the following commonalities between our work and the related work enhancing IDEs by means of program analysis and a more sophisticated information presentation:

- *Embedding information in familiar source views.* Most presented related work (Seesoft [EICK 92], Microprints [DUCA 05a], Fluid source code views [DESM 06]) seamlessly embed additional information in the existing, familiar IDE perspectives, which makes it easier for developers to learn and use these tools and techniques in their daily work. In all our work, we also aim at embedding our new tools and techniques deeply in the existing IDE views and perspectives. Our proposals, however, exploit more information than the tools presented in this section, for instance also development activity, historical, or runtime information.

- *Heat coloring.* Seesoft [EICK 92] and to some degree also Microprints [DUCA 05a] use a heat coloring metaphor to highlight important artifacts to draw a developer's attention to them. We follow the same idea in *HeatMaps* and also *Senseo*.

- *Linking related artifacts.* Fluid source code views [DESM 06] link artifacts statically related to the currently selected artifact to improve system understanding by connecting conceptually related parts of the software space. *Hermion*, *Senseo*, and *CollView* also link related elements, however, they dynamically analyze the system to discover these links.

2.1.2 Source History Analysis

Other approaches mine software repositories to identify entities related or coupled, for instance by exploring how source artifacts frequently changed together in the past. Mining the source history can reveal relationships between source artifacts that are visible neither in the static software structure nor in its dynamic behavior such as dependencies between source elements and configuration files.

Shirabad *et al.* [SHIR 03] use information about artifacts with common change patterns to recommend developers to also change the related entities when working on an artifact. This approach works at the granularity of files. Thus it cannot directly relate source artifacts such as classes or even methods to each other. However, to achieve our goal of identifying dynamic communication between source elements, we need to cover packages, classes, interfaces, or methods, in particular when dealing with non-file based languages such as Smalltalk. The approach of Shirabad *et al.* [SHIR 03] is promising, though, as it exploits both evolutionary information from software configuration management (SCM) systems and information from bug tracking systems. Co-update relations between artifacts, that is, artifacts that need to change together, are determined using machine learning techniques. This proposal is not accompanied with a tool integrated in an IDE, thus it does per se not solve any IDE problem. Furthermore, the quality of the co-update recommendations is highly dependent on the SCM system used [SHIR 03], thus we cannot exploit this approach in our work as we want to be independent of any SCM system.

Ying *et al.* [YING 04] propose an approach to mining change history by identifying dependencies between source elements. This approach complements static and dynamic analysis which cannot always identify all code relevant for a change, in particular for software systems using multiple programming languages. This proposal differs from the work of Shirabad *et al.* [SHIR 03] as it only relates source artifacts that changed together repeatedly, which improves the correctness of the recommendations [YING 04]. This approach, however, is also not integrated in an IDE and works at the granularity of files, too.

ROSE [ZIMM 04a] is very similar to the approach of Ying *et al.* [YING 04] and basically just differs in the mining algorithm used. While the ROSE approach uses association rule mining [ZIMM 04a], Ying *et al.* [YING 04] opted for frequent pattern mining. The predictability of the recommendations measured in precision and recall of the two approaches is similar [YING 04]. ROSE is available as an Eclipse plugin (cf. Figure 2.9).

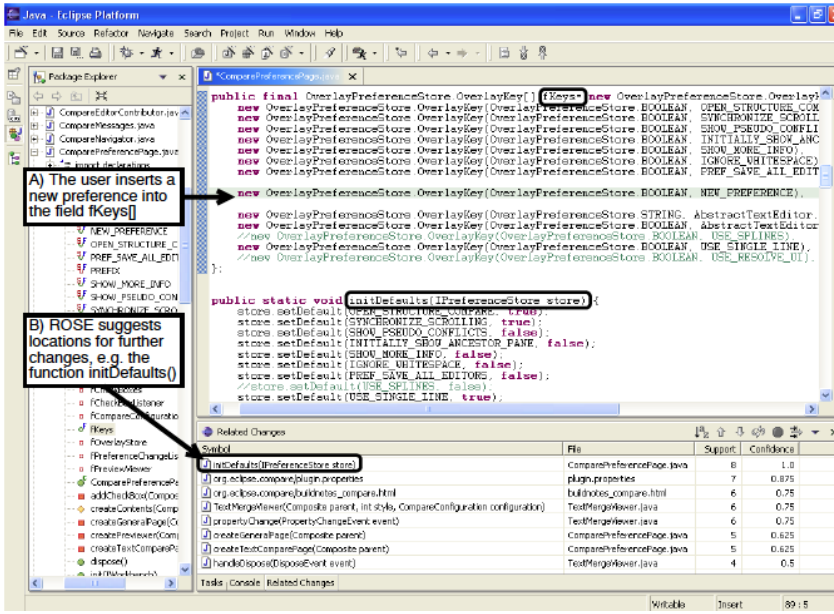

Figure 2.9: ROSE's suggestion view (lower right) integrated into Eclipse.

Hipikat [CUBR 03] recommends artifacts that are relevant for a specific task by exploiting different sources of information, namely the bug database as well as the source repository of the project, and even emails from newsgroups or other message archives are analyzed. Hipikat is available as an Eclipse plugin (cf. Figure 2.10) which particularly supports developers unfamiliar with a software system in learning from the implicit "project memory" stored in the system's change history [CUBR 03]. However, Hipikat requires a similar task to have been performed on a system in the past to provide specific recommendations of relevant artifacts for the current task-at-hand [CUBR 03]. Furthermore, an important prerequisite for Hipikat is a formal textual definition of the modification task (*e.g.* in Bugzilla) [CUBR 03]. From this definition, Hipikat starts to query the different data sources for similar tasks to suggest to developers what source element might be of interest for the task-at-hand. However, in practice there are often no formal definitions of tasks available, and past tasks might not be similar or not be specified accurately enough to be able to relate them to current tasks. The requirements Hipikat imposes on the source and change history and the task management are too severe for our goals.

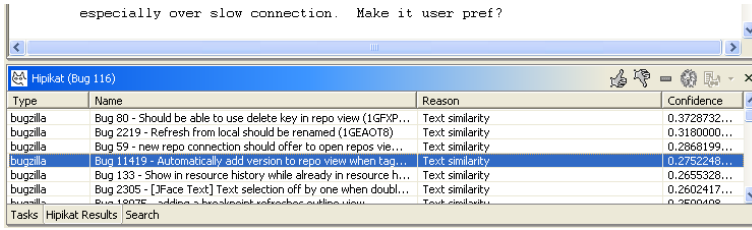

Figure 2.10: Hipikat's results view showing tasks similar to the currently performed task as specified in a task report.

The IDE problems addressed by the four presented mining source history approaches [SHIR 03, YING 04, ZIMM 04a, CUBR 03] are presented in Figure 2.11. Note that only ROSE and Hipikat have actually been implemented as IDE enhancements. The other approaches could probably be integrated easily into an IDE such as Eclipse.

Problem tackled / Approach	Information overload	No overview	No context, task support	Quality assessment support poor	Dist-ributed artifacts	Colla-boration hidden	Execution paths hidden	Imprecise static source code	Features hidden in code
Hipikat, ROSE, and others	Quicker navigation				If artifacts changed in tandem	If artifacts changed in tandem			

Figure 2.11: ROSE, Hipikat, and other mining approaches tackle the problem of related but distributed artifacts whose collaboration is hidden in IDEs. Furthermore, they also mitigate the information overload problem as related artifacts can be easily navigated using the recommendation lists.

Other researchers such as Xie *et al.* [XIE 06] show a complete picture of evolutionary data extracted from software repositories to augment the understanding of a system's evolution, but these visualizations are usually very large and outside of the IDE and thus of limited use while working with the static system structure.

Summary. The following commonalities exist between the related work enhancing IDEs by means of source history analysis and our own work:

- *Exploiting evolutionary information.* ROSE [ZIMM 04a], Hipikat [CUBR 03], and other previously presented proposals discovered the usefulness of evolutionary information to recommend artifacts developers should consider to study in order to perform a certain task. We also take evolutionary information into account in our

work on *HeatMaps* and *SmartGroups*, but combine this information with other sources of information such as development activity or dynamic information to achieve better results.

- *Recommending relevant entities.* These related proposals also give evidence that recommending relevant entities indeed helps developers to better understand a system or to more efficiently accomplish software maintenance tasks. Thus, we adopt this idea in *SmartGroups* by categorizing entities that are likely to be relevant for the current software maintenance task.

2.1.3 Developer Activity Analysis

Other techniques analyze development activities (usually investigation (navigation) and modification actions performed in the IDE) instead of mining software repositories to identify relations between source artifacts.

FEAT [ROBI 03a] applies a concern graph to visualize scattered but conceptually related code elements together in order to identify and navigate elements relevant for a particular concern. Recent versions of FEAT are able to automatically infer the source entities related to particular concerns [ROBI 03b]. Robillard *et al.* define a concern as "anything a stakeholder may want to consider as a conceptual unit, including features, nonfunctional requirements, and design idioms" [ROBI 07]. Usually the source code implementing a concern is not encapsulated in a single source entity, but is instead scattered over the entire system [ROBI 07]. To determine the entities participating in a concern, FEAT analyzes system investigation activities performed by the developer in the IDE [ROBI 03b]. FEAT is able to identify relevant concerns from a transcript of investigation activities with a manageable level of noise [ROBI 03b]. The resulting concern graph presented in the FEAT Eclipse plugin (see Figure 2.12) supports developers performing maintenance tasks involving identified concerns [ROBI 03b].

From the IDE problems identified in Section 1.1.2 FEAT primarily addresses the lack of overview in IDEs and the fact that artifacts relevant for a concern are distributed over the entire source space. FEAT also mitigates the problem of information overload, hidden collaborations, and missing representation of features in IDEs. However, as FEAT does not exploit dynamic information, the last two problems are only loosely addressed, basically when developers have navigated the system in the past in such a way that they came across artifacts whose collaboration is only dynamically visible or that participate in the same feature.

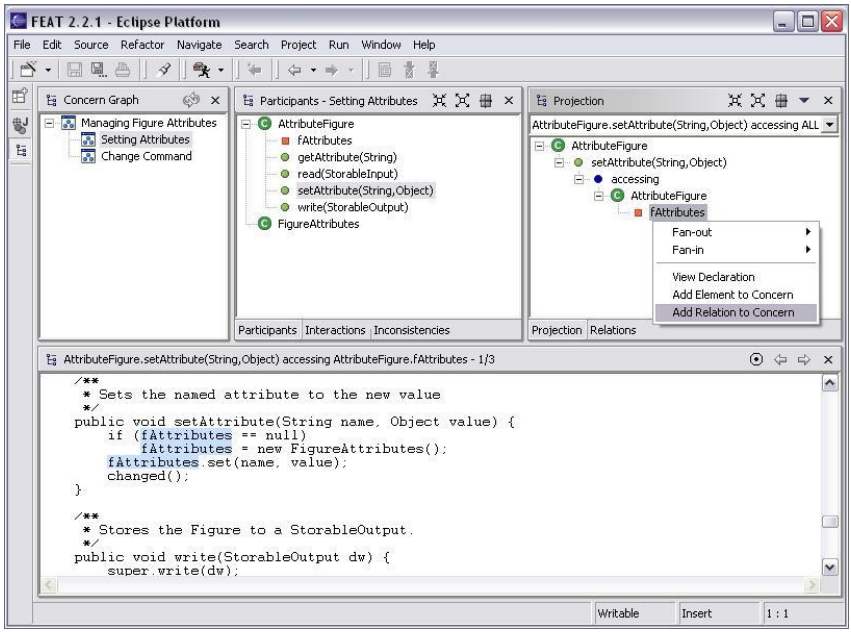

Figure 2.12: A concern representation in FEAT integrated in Eclipse.

Otherwise such collaboration is not made explicit by FEAT. As the authors report, FEAT's concern identification algorithm is in general heavily dependent on how organized the analyzed investigation activities are [ROBI 03b, ROBI 07]. Disorganized investigation sessions yield vague, incomplete and often useless concern graphs [ROBI 03b]. Thus the FEAT approach is not very robust and not decently usable when only having available development sessions from developers unfamiliar with the system under study. Furthermore, the FEAT approach requires developers to manually validate the proposed concerns by rejecting false positives, that is, concerns wrongly identified. Figure 2.13 lists the IDE problems partially addressed by FEAT.

NavTracks [SING 05] exploits navigation history to recommend files related to the file the developer is currently looking at. Next to a source file, NavTracks shows in Eclipse a view listing related files, as shown in Figure 2.14. This related files list is ranked by the recency of navigation; the higher in the list the more recently this file has been navigated. This approach works at the granularity of files, hence does not take into account specific methods or classes.

Problem tackled / Approach	Information overload	No overview	No context, task support	Quality assessment support poor	Dist-ributed artifacts	Colla-boration hidden	Execution paths hidden	Imprecise static source code	Features hidden in code
FEAT	Limited to concern graphs	Overview of concerns improved			If have been navigated in tandem	If have been navigated in tandem			If entities have been navigated

Figure 2.13: FEAT tackles the problems of missing overview and information overload. If recorded navigation activities are accurate, FEAT can reveal hidden dependencies between distributed source artifacts relevant for specific features.

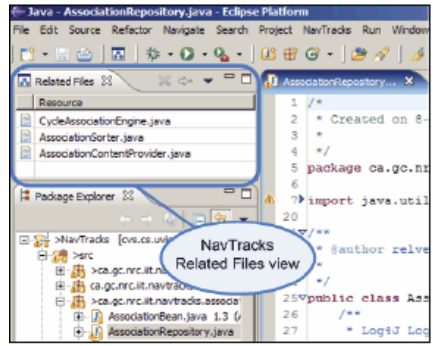

Figure 2.14: NavTracks' related files view integrated in Eclipse.

Similar to FEAT, NavTracks also tackles the problems of distributed artifacts and hidden dependencies between them that are not easily discoverable and navigable in IDEs. Additionally, NavTracks also mitigates information overload as developers can directly navigate related artifacts by consulting the recommendation list appearing for each selected source element. This often saves developers from having to search in the large software space for related artifacts. The IDE problems addressed by NavTracks are listed in Figure 2.15.

However, we question the quality of the recommendation list NavTracks provides. NavTracks only takes into account one single data source, namely the recency of browsing in the navigation history, to assess the relatedness of artifacts. Other sources or even combinations of different sources, such as combining frequency and recency of modification and navigation of entities, could lead to much better results. Hence it is questionable whether NavTracks is able to correctly identify related artifacts. As for FEAT, the performance of NavTracks is also highly dependent on the quality and nature of the recorded navigation history, thus a correct

Problem tackled / Approach	Information overload	No overview	No context, task support	Quality assessment support poor	Distributed artifacts	Collaboration hidden	Execution paths hidden	Imprecise static source code	Features hidden in code
NavTracks	Quicker navigation				If navigated in tandem	If navigated in tandem			

Figure 2.15: NavTracks mitigates the information overload problem as related entities can be quickly navigated with the recommendation list, which also contains distributed artifacts whose collaboration is otherwise not explicit in the IDE.

identification of dynamic dependencies between distant source artifacts is certainly not possible in all cases.

Furthermore, a recommendation list helps little to obtain an overview of the whole system; the developer just sees a list of artifacts possibly related to a specific artifact, but does not see all interesting entities in a "big picture" view. These recommendations are always relative to a selected artifact, that is, dependent on what the developer has currently selected, thus it is not easy to identify all artifacts related for a given task. The model of NavTracks does not consider the notion of tasks, thus related entities are recommended independently of a particular context, task, or concern. The exchange or the exploration of data sources recorded by different developers is also not supported with NavTracks as its model is built on the client side in this specific environment.

NavTracks has only been evaluated by analyzing the recorded navigation activities of three developers. The correctness of NavTracks was determined by checking whether the file browsed next in the recorded history appeared in the recommendation list (hit) or not (miss). Correctness equals to number of hits divided by number of navigation events (hits plus misses). The average correctness for all three developers was below 30%. We consider this evaluation to be rather weak as the variance between the three developers was high, thus the number of testimonials should be much higher to achieve a certain degree of power and significance.

Team Tracks [DeLi 05a] follows a similar approach as NavTracks, but also exploits the code navigation patterns of team members to relate source elements, thus fixing the issue of NavTracks not being able to explore navigation data from more than one developer. Besides providing a list of related artifacts when viewing a particular source element, Team Tracks also provides a favorite class view which hides classes less frequently navigated by all team members. DeLine *et al.* [DeLi 05a]

claim that sharing team navigation data can further improve the quality of the related item lists and thus eventually better improve program comprehension.

Concerning the advantages and shortcomings of Team Tracks, in particular of its related item view, the same arguments hold as mentioned for NavTracks.

Mylyn (formerly known as Mylar) [KERS 05, KERS 06] computes a degree-of-interest value for each source artifact based on the historical selection or modification of the artifact. The background color of the artifacts highlights their relative degree-of-interest in the context of the current task — interesting entities are assigned a "hot" color. In Mylyn the information used to compute the interest value is relatively simple: selecting and editing an artifact increases the interest; if no further event occurs the interest decreases over time. Mylyn has been validated by means of a field study [KERS 06] in which 16 subjects provided decent longitudinal data that could be analyzed. The results showed that Mylyn significantly improves the edit ratio, that is, the ratio between number of modification and navigation activities. Thus, when using Mylyn developers perform fewer navigation actions to locate the entities to be modified to, for instance, correct a defect.

Mylyn addresses basically the same IDE problems as NavTracks, that is, information overload and scattered, distributed artifacts with hidden collaboration between them (see Figure 2.17). Additionally, Mylyn also provides a representation of context by highlighting task-relevant source entities, and can even represent features if the recorded development activities allow for their appropriate identification. Highlighting of relevant source artifacts also enhances the overview of the system, at least for particular tasks represented with Mylyn.

The degree-of-interest model contributed by Mylyn is likely to yield more precise and accurate results concerning the identification of related entities than NavTracks' algorithms. However, developers cannot influence how the interest value is computed as the algorithms are fixed. The interest in an artifact is highly dependent on the nature of a task though. Developers are probably interested in different artifacts when correcting a defect than when implementing a new feature. Mylyn, however, just exploits one source of information, development history (navigation and modification activities), and the model analyzing this data does not adapt to the nature of the task. For many tasks additional data sources such as evolutionary or dynamic information could, however, predict the interest values of artifacts much more accurately [RÖTH 09c].

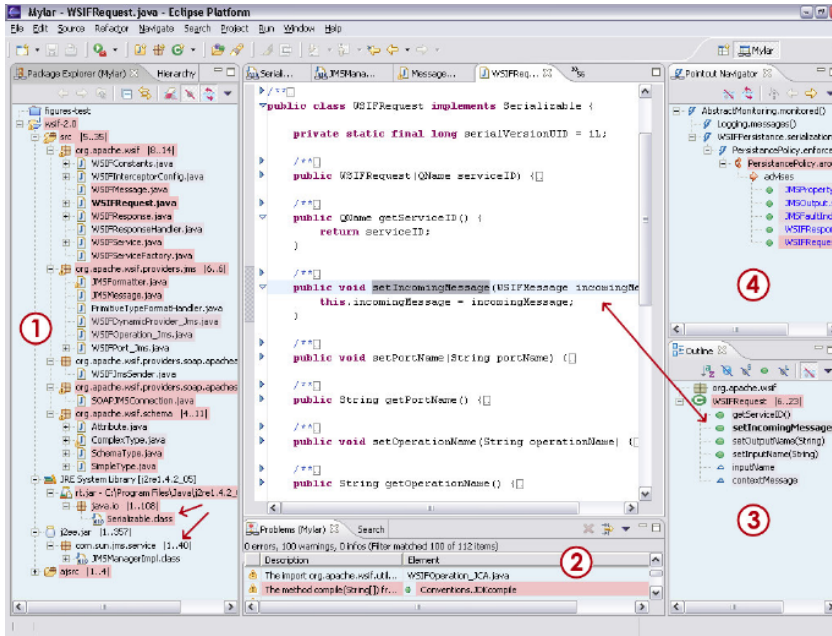

Figure 2.16: The different views provided by Mylyn in Eclipse.

As professional developers reported to us in informal discussions, Mylyn's contribution to the reduction of the information overload and the improvement of system overview is limited for large systems. For such systems, Mylyn's task views grow crowded with many artifacts after a while, thus the overview is hampered also in these task views. This is a hint that Mylyn's approach of task identification does not scale well. Thus Mylyn does not yet completely solve the problems of IDEs such as information overload, lack of overview and explicit context representation.

Summary. Our work has the following points in common with the previously presented related work:

- *Exploiting development activities.* All approaches presented in this section give evidence that recorded development activities are an appropriate source of information to either recommend to developers entities they should consider for particular tasks or to represent a working context, that is, a set of entities being relevant in a specific context, for instance when having to perform a certain task such as

Problem tackled / Approach	Information overload	No overview	No context, task support	Quality assessment support poor	Dist-ributed artifacts	Colla-boration hidden	Execution paths hidden	Imprecise static source code	Features hidden in code
Mylyn	Highlighting of relevant entities	Overview of tasks improved	Limited support for tasks		If have been navigated in tandem	If have been navigated in tandem			If entities have been navigated

Figure 2.17: Mylyn highlights interesting artifacts to mitigate the information overload, identifies task-relevant entities, and, dependent on the quality of the development activities, reveals hidden collaboration between artifacts or identifies entities used in particular features.

fixing a defect. Thus, *HeatMaps* and *SmartGroups* both also exploit development activity information to identify artifacts relevant for specific software maintenance tasks, but combine this information with other sources such as dynamic or historical information to obtain better results.

- *Representing task-relevant context and concerns.* FEAT [ROBI 03a] and Mylyn [KERS 05, KERS 06] propose to explicitly represent software concerns and task-relevant entities. We strive for a similar goal in *SmartGroups* but exploit more information sources, offer a more flexible model to compute task-relevant entities, and restrict the number of identified entities in order to not overload developers with too many, eventually unrelated entities.

- *Recommending relevant entities.* NavTracks [SING 05] and Team Tracks [DELI 05a], similar to ROSE [ZIMM 04a] or Hipikat [CUBR 03], show that recommending relevant entities to developers, *e.g.* entities developers should also modify to complete a defect correction task, is indeed a practicable aid for developers. Hence, we follow the same principle in *SmartGroups* which, however, provide general lists of task-relevant entities that are only dependent on the type of task being performed, but not on the currently selected entity as in NavTracks or Team Tracks.

2.1.4 Debugging, Profiling

Many IDEs also provide support for debugging and profiling. In this section we discuss some recent and advanced debuggers and profilers available in IDEs.

Whyline [KO 04] is a prototype interrogative debugging interface for the Alice programming environment. Whyline enables developers to ask *why did* and *why did not* kind of questions about runtime failures. Alice[2] is an event-based language to simplify the creation of interactive 3D worlds [KO 04]. To create code, developers drag and drop tiles to the code area and choose parameters from popup menus, similar as in the Scratch environment [MALO 04]. This interactive and visual way of programming is extended by Whyline to allow developers to ask questions about objects being part of the world developed in Alice, for instance questions such as why a particular button was not activated at runtime or why a figure did not change its skin. To concretely ask such questions, the developer selects a particular object shown by Alice and scans the property changes that could have happened for this object during the execution of the system [KO 04]. For each selected property, the code that caused the property change is highlighted. Whyline analyzes the runtime actions to also reveal which code has not been executed that could have changed the property, and why not, for instance because a condition was false. Consequently, Whyline exposes hidden dependencies between actions and data that are otherwise hard to determine [KO 04].

Problem tackled / Approach	Information overload	No overview	No context, task support	Quality assessment support poor	Dist-ributed artifacts	Colla-boration hidden	Execution paths hidden	Imprecise static source code	Features hidden in code
Whyline						Occasion-lly for specific executions	Only for visual objects	Only for visual objects	Limited support

Figure 2.18: Whyline improves the understanding of static source code, execution flow, and features. Hidden collaborations can be also spotted in some cases.

Whyline tackles the problem of unclear static source code, hidden execution paths, and even, to some degree, of software features hidden in code since Whyline allows developers to map program behavior and features to particular code statements. Similarly, Whyline also reveals hidden collaborations between artifacts in some cases. Figure 2.18 summarizes the IDE problems tackled by Whyline.

While such an interrogative debugging approach directly integrated in the IDE is interesting, it is a long way to go to support this approach in complex, object-oriented languages and environments such as Java and Eclipse [KO 04]. The sheer number of possible questions in a Java program, the issue of efficiently analyzing the complete execution history of the program, or the presentation of a dynamic slice in the IDE are

[2] http://www.alice.org

reasons why an adaptation of the Whyline approach is not yet practical for the context of Java.

Compass [LIEN 09] is a back-in-time debugger available for Smalltalk which also allows developers to navigate back-in-time through all the code that has touched a particular object. Compass addresses the problem that the cause of many bugs is not visible in the execution stacks provided by conventional debuggers [LIEN 09]. While typical back-in-time debuggers such as TOD [POTH 07] enable programmers to step back through earlier states than the current state of the program, they cannot reveal from where a particular object relevant for a bug comes [LIEN 09]. For this reason, Compass also tracks the flow of objects (cf. Figure 2.19), for instance to detect the method which stored a particular value in an object.

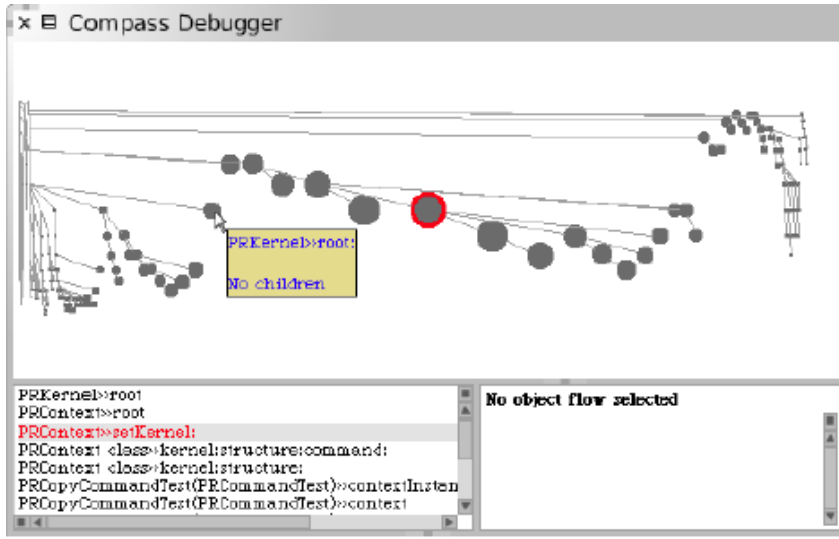

Figure 2.19: The method trace view of Compass visualizes the entire runtime control flow as a tree of nodes in a fisheye view. A node represents a method execution. The call stack below the method trace view focuses on a single slice of the trace.

Tracking the object flow supports developers in correcting hard to fix defects, but can also reveal hidden collaborations between distant source artifacts or help programmers understanding features. As any debugger, Compass also improves the understanding of executions paths and source code with abstract static types or no static types at all (cf. Figure 2.20).

However, the limitation of debuggers is, first, that they focus on a specific system execution and thus cannot provide general information such as which source artifacts are used in a feature, which are the different methods invoked at a polymorphic call site, or which types of objects are stored in a particular variable. Second, debuggers do not integrate the dynamic information in the static source views. Debugging information is volatile in the sense that the reified dynamic information is bound to a specific debugging session. Developers can investigate the execution stack or, thanks to Compass, also the object flow, or are able to manually explore collaborating artifacts or entities participating in a feature. But this information stems from a snapshot of the program's runtime and is only valid for one particular execution. This reified information is not fed back to the IDE for persistent access.

Furthermore, debuggers do not provide an explicit representation of artifacts' collaboration or their participation in particular features. It is for example not possible to select an artifact in the debugger to see all the artifacts with which it collaborates in this particular execution. While a debugger, including Compass, is great to investigate how particular source entities communicate in a specific execution, it fails to reveal a general, "big picture" view of a system, for instance how source artifacts collaborate in general, that is, in many different executions and software features.

Problem tackled / Approach	Information overload	No overview	No context, task support	Quality assessment support poor	Dist-ributed artifacts	Colla-boration hidden	Execution paths hidden	Imprecise static source code	Features hidden in code
Compass					For specific executions	For specific executions	For specific executions	For specific executions	

Figure 2.20: Compass reveals hidden dependencies between distant source artifacts and improves understanding of static source code and execution flow in specific system executions.

JFluid [DMIT 04b] is a Java profiler integrated in the NetBeans IDE. The biggest advantage of JFluid is its efficiency; its profiling overhead is very small compared to other profilers (cf. Section 2.2.2) [DMIT 04b]. JFluid collects two kinds of CPU profiling data: the calling context tree (that is, basically the method invocation tree) and gross execution time for single code regions. JFluid provides a simple interface to show this data in the NetBeans IDE. Thus, JFluid aims at supporting developers in locating performance bottlenecks in their code. JFluid does not directly mitigate any IDE problems besides, to some extent, improving the support for

software quality assessment in IDEs as this tool pinpoints entities being slow to execute.

Summary. Like Whyline [KO 04], Compass [LIEN 09], and JFluid [DMIT 04b], our work also exploits dynamic information to improve system understanding. However, while these approaches present volatile information about a specific system execution, we want to integrate with *Hermion* and *Senseo* aggregated dynamic information in the IDE that is permanently available and embedded in the traditional source code views. *CollView* and *FeatureEnv* visualize dynamic information in permanently accessible visualizations integrated in the IDE. This dynamic information is usually also aggregated over various system or feature executions.

2.1.5 Querying

JQuery [JANZ 03] is a code browsing tool implemented on top of an expressive logic query language. It combines a hierarchical browser with the flexibility of a query tool. In a single integrated view in Eclipse, JQuery provides an explicit representation of the exploration path followed by the developer [JANZ 03]. Query results are shown in a tree which only serves as a starting point for the exploration process [JANZ 03]. Each tree node can be further expanded to explore entities connected to the selected node through relationships such as being invoked by or invoking this particular node (*e.g.* a method). JQuery uses purely static information to find the results for such queries. The query language used by JQuery, TyRuBa[3], has the expressive power to also formulate complex queries, however, such queries are not likely to be used by developers, thus JQuery also provides pre-defined queries, saving developers from formulating the queries themselves [JANZ 03].

According to the authors, JQuery is supposed to prevent developers from getting lost by making relationships between scattered code elements more tangible [JANZ 03]. JQuery should make the navigation of crosscutting concerns easier by reducing the need for disorienting view switches and by explicitly representing the exploration process in terms of exploration paths in a tree integrated in Eclipse [JANZ 03]. Thus, JQuery addresses the problem of being overloaded with information in the IDE and of hidden relationships between scattered and distributed artifacts. To some degree, JQuery also improves the overview of the system and in particular of the exploration process (cf. Figure 2.22). However, as JQuery exploits just static relationships between source artifacts, it cannot address the problems we discussed in Section 1.1.2 concerning purely

[3] http://tyruba.sourceforge.net

Figure 2.21: An example of JQuery showing in Eclipse an exploration process tree starting with the results of a query.

Approach \ Problem tackled	Information overload	No overview	No context, task support	Quality assessment support poor	Dist-ributed artifacts	Colla-boration hidden	Execution paths hidden	Imprecise static source code	Features hidden in code
JQuery	Fewer windows	Concerns explicit			Just static relation-ships	Just static relation-ships			

Figure 2.22: JQuery reduces information overload in IDEs by explicitly representing concerns, thus relevant artifacts can be studied in a single perspective, which also improves the overview. Hidden collaboration between distributed artifacts is determined purely by static analysis.

dynamic collaborations. Furthermore, the information overload and lack of overview problem is still present when using JQuery, in particular as the tree representing the exploration process very quickly grows large. Furthermore, JQuery does not take into account any task-specific information. However, an exploration process to fix a defect is likely to differ from one concerned with the implementation of a new feature.

Ferret [DE A 08] recognizes the conceptual relation between static and dynamic aspects of software systems by integrating a query tool into Eclipse to allow developers to execute conceptual queries about source artifacts directly in the IDE. An example of such a query is "callers of method x". Ferret focuses on querying static information, but is also able to take into account dynamic and evolutionary information to obtain more precise results [DE A 08]. Ferret implements 36 conceptual queries of which five also consider dynamic information. When Ferret is invoked

for a particular source artifact, it computes and displays the results of all queries appropriate for that artifact. Developers cannot formulate their own query, Ferret pre-defines fixed conceptual queries in the Ferret view as shown in Figure 2.23. Queries can be cascaded, that is, query results are used as sources to formulate new queries [DE A 08]. Such cascaded searches are visualized in a tree. Ferret was validated in a two-day field study with four professional software developers. The study subjects used nearly all 36 conceptual queries provided to them and considered the results as useful [DE A 08].

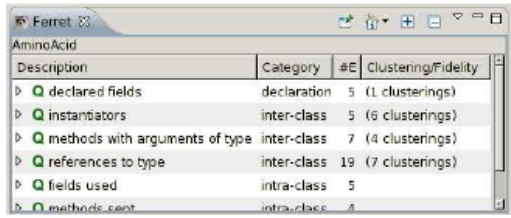

Figure 2.23: Ferret's query results view integrated in Eclipse.

The authors claim that Ferret particularly addresses the information overload in IDEs and the lack of focus and overview [DE A 08]. Additionally, we think that Ferret also contributes to make collaboration between distant artifacts visible by supporting queries revealing, for instance, callers of a method. Even on a low source code level, Ferret augments the understanding of unclear execution flow and static source code by identifying methods actually invoked at runtime. As Ferret uses a list to show the results of all queries appropriate for the artifact in question, developers have to spend quite some time skimming through this list to find useful information. The computation of these results also takes considerable time. The different IDE problems mitigated by Ferret are summarized in Figure 2.24.

While having answers to Ferret's queries is useful to developers in many situations, we are skeptical whether the way Ferret integrates the query results actually mitigates information overload, as first of all Ferret integrates an additional view with plenty of information, thus rather increases the amount of information presented by the IDE. Ferret clearly makes a contribution to identifying related artifacts, but does not solve the problem of being disoriented and not having a task-dependent context. Ferret does not take into account development context or tasks. The results it shows are only dependent on the currently selected artifact, but are not filtered or otherwise processed dependent on the current task. Furthermore, the dynamic information exploited by Ferret is very limited. It basically only reasons about method invocations, thus no runtime type

Problem tackled / Approach	Information overload	No overview	No context, task support	Quality assessment support poor	Dist-ributed artifacts	Colla-boration hidden	Execution paths hidden	Imprecise static source code	Features hidden in code
Ferret	Limited as query view is over-loaded	Limited as query view is over-loaded			Focus on currently selected artifact	Static, only method invocations dynamic	Static, only method invocations dynamic	Static, only method invocations dynamic	

Figure 2.24: By providing a dedicated but often overloaded query view, Ferret improves to some degree information overload and overview in the IDE. For the currently selected artifact, related artifacts are revealed based on static and dynamic analysis. However, only method invocations are dynamically analyzed, thus support for the understanding of execution flow, static source code, and dynamic collaborations is limited.

or memory consumption information is provided. Ferret's support to comprehend source code hard to understand due to the use of abstract types or late-binding is hence limited. These concerns are subsumed in Figure 2.24.

Summary. Our work shares the following points with JQuery [JANZ 03] and Ferret [DE A 08]:

- *Relating scattered code.* JQuery [JANZ 03] and also Ferret [DE A 08] relate scattered and distributed code by allowing developers to formulate queries whose results reveal artifacts that are conceptually related but statically distributed. With *Hermion*, *Senseo*, *CollView*, or *FeatureEnv* we pursue the same goal, but exploit behavioral information to relate distributed code instead of exploiting recorded navigation activities (JQuery) or static program information (Ferret). Moreover, our proposals embed this information directly in the source perspectives, thus saving developers from the need to formulate queries.
- *Static and dynamic information combined.* Ferret [DE A 08] is capable of also taking into account dynamic information for some of their pre-defined queries. In *Hermion* and *Senseo* we also combine the static perspective with the dynamic view on a system to improve program comprehension. We, however, use more dynamic information than just method invocation, for instance also type information or complexity information such as number of objects created, etc.

Other approaches combine querying software structure with visualizations. GraphLog [CONS 92] for instance aims at simplifying complex

relationships among software artifacts by translating them into a graph. Developers can interact with this graph to visually formulate queries to find patterns in the relationships between source elements, such as which classes invoked a particular method.

	Approach / Problem tackled	Seesoft	Micro-prints	Fluid source code views	CodeSonar	Hipikat, ROSE, and others	FEAT	NavTracks	Mylyn	Whyline	Compass	JQuery	Ferret
Overloaded views	Information Overload	Only on source code level	Only on method and single class level	Fewer windows		Quicker navigation	Limited to concern graphs	Quicker navigation	Highlighting of relevant entities			Fewer windows	Limited as query view is over-loaded
	No overview	Only on source code level	Only on method and single class level	Limited support	Only for classes and their static relations		Overview of concerns improved		Overview of tasks improved			Concerns explicit	Limited as query view is over-loaded
	No context, task support								Limited support for tasks				
I	Quality assessment support poor				For class level quality defects								
Narrow focus on static views	Distributed artifacts			Statically linked methods		If artifacts changed in tandem	If have been navigated in tandem	If have been navigated in tandem	If have been navigated in tandem		For specific executions	Just static relation-ships	Focus on currently selected artifact
	Collaboration hidden					If artifacts changed in tandem	If have been navigated in tandem	If have been navigated in tandem	If have been navigated in tandem	Occasion-lly for specific executions	For specific executions	Just static relation-ships	Static, method invocations
	Execution paths hidden									Only for visual objects	For specific executions		Static, method invocations
	Imprecise static source code									Only for visual objects	For specific executions		Static, method invocations
	Features hidden in code						If entities have been navigated		If entities have been navigated	Limited support			
	Static Analysis	✔	✔	✔	✔								✔
	Dynamic Analysis	✔								✔	✔		✔
	Source History Analysis					✔							✔
	Development Activity Analysis	✔					✔	✔	✔			✔	
	Debugging, Profiling									✔	✔		
	Querying											✔	✔

Figure 2.25: Summary of the different IDE problems tackled by the presented related works. All problems are mitigated, but not any of them thoroughly.

2.1.6 Conclusions

To conclude this section about related work in the context of development environments we analyze which problems and issues of IDEs these related approaches address, and to which degree. Figure 2.25 gives an overview of the IDE problems each presented proposal tackles. This table shows that all problems have been partially addressed by at least one approach. However, even all approaches combined are not able to address any of the problems completely.

Considering the information overload problem, for instance, some approaches just reduce or better display information on a source code level (Seesoft, Microprints), other approaches require the developer to open slightly fewer windows (Fluid source code views, JQuery), yet other proposals add shortcuts to ease identifying and navigating to important artifacts (Hipikat, NavTracks, Ferret, but also Mylyn or FEAT). But none of these approaches is able to significantly reduce the number of entities or windows when navigating the entire software space. Mylyn comes close to this goal, but fails to scale to large systems and its means to associate artifacts to specific tasks does not yield optimal results.

The other important problem of IDEs, the narrow focus on static views, in particular their missing representation of collaboration between distant artifacts, is not well addressed by the considered proposals. Either the proposals just statically link the artifacts and thus suffer from the imprecision of static analysis (Fluid source code views, JQuery, Ferret), depend on whether the collaborating artifacts have previously been changed or navigated together (Hipikat, FEAT, NavTracks, Mylyn), or they focus on specific system executions and do thus not provide general information (Whyline, Compass). Hence the current research in the context of development environments neither accurately nor completely tackles the two main problems of IDEs we identified (cf. Section 1.1.2), namely information overload and the narrow focus of IDEs on static software structure, including all subsequent sub-problems.

2.2 Software Analysis and Visualization

In this section, we briefly discuss several proposals to statically or dynamically analyze software systems and to visually present analysis results. These analyses and visualizations are usually not integrated in IDEs, but provided in separate tools. We mainly focus on approaches to dynamically analyze software systems as we want to extend IDEs to integrate dynamic information. We thus carefully study related work on dynamic analysis and report on how to use and extend existing work to reach our goal of enriching IDEs with dynamic information.

Gathering information using different analysis techniques is of limited use if this information is not well presented to developers. Graphical representations of software and analyses results have long been accepted as an appropriate comprehension aid [STAS 98]. The work of Maletic *et al.* has provided important guidelines for motivating and defining visualizations. In their work they defined levels of interest and the criteria

of effectiveness and expressiveness of software visualization [MALE 02]. Most visualizations presented hereafter follow these guidelines.

2.2.1 Means to Present Static or Historical Information

Moose [NIER 05] is a software analysis platform encompassing various software visualizations such as different polymetric views [LANZ 03] to support reverse engineering tasks. Other tools or environments visualizing static information are for instance Rigi [TILL 94], Hy+ [MEND 95], Dali [KAZM 99], or Tango [STAS 90]. These environments are standalone applications that usually do not integrate their services in any conventional development environment.

We study in the following in more detail several approaches that could easily be integrated in IDEs. All these approaches mainly tackle the problem of information overload and missing overview, some additionally help developers to understand the execution flow inside classes and methods or to identify collaborating artifacts.

Generalized fisheye views [FURN 86] are adapted source code views that provide a balance of local detail and global context by trading off importance against distance. The code segment near to the current focus point is shown in detail while only important lines of code are shown for segments further away [FURN 86]. Fisheye views thus improve the overview of source code, but cannot improve the understanding of a system as a whole. For object-oriented languages with rather short methods, fisheye views are less useful.

Polymetric Views [LANZ 03] are lightweight visualizations mapping different software metrics to two-dimensional nodes representing entities. Height, width, position, or color of a node can each express a particular metric value. Nodes are connected by edges representing relations between source entities such as invocation or inheritance. Polymetric views have been applied to many different visualizations such as the system complexity view [LANZ 03], class blueprints [DUCA 05b], or the condensed runtime information view [DUCA 04]. Polymetric views can serve a multitude of different purposes such as providing an overview of a system, of a group of source artifacts, or of single artifacts. Another purpose is to support the understanding of control flow or relations between artifacts. Usually, polymetric views show statically determined information, but as a general-purpose visualization they can easily be enhanced with dynamic information. In Chapter A, we elaborate on how

we integrated polymetric views based on static and dynamic information in the IDE.

Whorf [BRAD 92] is a software maintenance tool hyper-linking distributed but conceptually related artifacts and explicitly visualizing the relationships between such distributed artifacts. Whorf exploits different kinds of (static) relationships between artifacts, such as variable and function references or method call relationships [BRAD 92]. The visual representations of these different relationships are displayed in interactive and linked views. Whorf aids developers in navigating distributed and scattered code.

SHriMP [STOR 01] provides an interactive environment for navigating and browsing complex source spaces by using nested graphs to browse hierarchical relationships such as inheritance. SHriMP allows developers to zoom from high-level visualizations down to low level representations of source elements such as Javadoc documentation or method source code. The views of SHriMP provide links to navigate from source code elements to the corresponding nodes in the visualization. The source code is not editable in SHriMP and source elements are related based on static information only. SHriMP particular aims at providing an overview of the system and to help newcomers to a system to build mental models of its higher-level design and architectural concepts. To the best of our knowledge, SHriMP's views are currently not integrated in any IDE.

Software Terrain Maps [DELI 05b] is an interactive visualization of source code, similar to cartographic maps, which provides landmarks to keep a programmer oriented while navigating around. These maps are provided by a dedicated, stand-alone tool. Software Terrain Maps aim at reducing disorientation while navigating source code by activating the spatial memory of the programmers to stay oriented. The software system is modeled as a set of components (*e.g.* methods) whose size is mapped to accordingly sized tiles arranged in a Voronoi diagram [DELI 05b]. The differently sized shapes serve as memorable visual landmarks easing the orientation and navigation in such maps. Different color shades provide further orientation guide. Software Terrain Maps particularly improves the overview of a system and the identification of code previously browsed, provided that its visual representation is a landmark that can be easily identified.

CodeMap [KUHN 08] enhances the basic principles of Software Terrain Maps. CodeMap is integrated in an Eclipse view appearing next to

structural views such as the package explorer. Navigation in the map and the traditional Eclipse source tools are linked, search results or open files are highlighted in the current map. CodeMap provides a spatial and stable mental model of software projects to developers by mapping a system's structure and vocabulary on a cartographic view. Source artifacts are shown as hills, the distance between them represents lexical similarity and structural closeness, the elevations of hills map artifacts' size expressed in KLOC. CodeMap addresses the same problems as Software Terrain Maps. Its landmarks, however, are usually more pronounced and thus easier to locate in the map. As the positions of source elements remain stable over time, the orientation in the map and the overview it provides is better than in the case of Software Terrain Maps.

CodeCity [WETT 08] uses a city metaphor to represent software structure and evolution. Packages are represented as districts, classes as buildings, and methods as stories in a building. Positions of source entities are determined based on the size of the entities using a modified tree layout. CodeCity is built on top of Moose [NIER 05] and not integrated in any IDE. CodeCity particularly aims at providing an overview of a system. However, as source artifacts may change position over time, CodeCity is less useful during software maintenance and evolution; it can be considered as a pure analysis tool for a system to better understand its high level static structure.

Kumpel [JUNK 09] is an interactive visualization simplifying the analysis of source file histories by visualizing the complete evolution of the source code contained in a file in a single view. Each file revision is shown as a vertical bar in this view. In this bar, each chunk of added code is colored according to the corresponding developer. Modifications are represented as small dots which are also colored according to the developer who performed the modification. Kumpel allows us to quickly identify who changed which part of a file when and to which extent. Kumpel is not integrated in an IDE, and while providing a comprehensive view on a source file's history, it does not help developers gaining an overview of a system. Kumpel's views are themselves overloaded with information which renders their adoption in an IDE rather difficult.

Summary. To summarize the discussion of these different related works concerned with static analysis, we report on their issues and limitations.

- First of all, these approaches, except CodeMap, are not integrated in IDEs and thus cannot directly solve the identified IDE problems.

Even when these proposals would be integrated, the issues mentioned below still exist.

- The presented visualizations would be an additional view in the IDE, separated from the conventional source views such as package tree and source editor. This separation is likely to distract developers as switching between fundamentally different views (for instance, between a visualization and the source editor view) imposes a cognitive burden [EICK 92, ROBB 05]. This burden, however, is dependent on how well such a visualization can be embedded in the way how developers interact with the IDE.
- The so far presented visualizations are not able to represent a working context of task-relevant elements or dynamic collaboration between distant artifacts, two fundamental issues of IDEs we want to address.
- None of these approach is capable of enriching the conventional tools of IDEs that developers are familiar with. All approaches throw in a completely new and unfamiliar means looking at and navigating in the source space. Usually, these means are read-only, they cannot be used to actually modify the software system under study.

For all these reasons, we search further for works able to provide information to be integrated in the conventional source perspectives of IDEs and that can also gather and exploit runtime information.

2.2.2 Dynamic Analysis

We first report on existing work to gather runtime information and second on approaches to visualize this information. These visualizations are usually not embedded in IDEs.

Dynamic Information Gathering

In order to be able to augment an IDE's static source perspectives with dynamic information, we first formulate the following requirements on a dynamic data gathering technique before looking at some concrete proposals:

- *Reification of sub-method elements.* As our goal is to improve the understanding of unclear static source code, we need to be able to

gather runtime information about sub-method statements such as variable assignments or message sending.

- *Mapping dynamic information to static source elements.* We aim at mapping the collected dynamic information back to static source code. For instance if the invocation of a particular method is written several times in a method's source code, we must be able to identify which runtime argument types were used at which location in the code.
- *Selective data gathering.* Often we are not interested in dynamically analyzing all parts of a system. Thus we should be able to select the specific artifacts (packages, classes or methods) and even particular kinds of operations (*e.g.* message sending, variable accessing) we want to cover by dynamic analysis.
- *Complete and accurate information.* Dynamic information to be integrated in the IDE needs to be accurate and complete in the sense that information has to be available for all the code executed in the recorded runs of the system under study. Note that dynamic analysis is normally not able to completely cover all system behavior [BALL 99], as usually only specific system features are executed and dynamically analyzed. But for these features being analyzed we expect to gather complete dynamic information with respect to the information to be integrated in the IDE.
- *Efficiency.* Analyzing the dynamics of software systems is considered to be slow, because huge amounts of data are generated when executing software systems. As the IDE should show dynamic information immediately after a system's execution, a tremendous slowdown of the subject system to acquire its execution data is not tolerable.
- *Extensibility.* The data gathering approach should be easy to extend, for instance to conveniently be able to integrate more or different kind of dynamic information in the IDE.

Dynamic analyses are usually based on tracing mechanisms. Tracing traditionally focuses on capturing a method call tree, but existing approaches usually do not bridge the gap between dynamic behavior and the static structure of a program [HAMO 04, DUNS 00, WILD 92]. Thus, Löwe *et al.* [LÖWE 01] merged information from static analysis with information from dynamic analysis to generate visualizations. Zaidman *et al.* [ZAID 05] or Hamou-Lhadj *et al.* [HAMO 05] mined static entities in dynamic tracing data to, for instance, reveal key classes forming good starting points for further analysis.

Many of today's tracing tools such as those based on method wrappers [BRAN 98] implement techniques allowing selective instrumentation of the source code based on criteria such as package boundary or on individual selection of methods to limit the amount of gathered data and the collecting overhead. However, only a few tracing techniques such as that described in the work of Ducasse *et al.* [DUCA 06] consider sub-method elements such as variable assignments or message sending. Often techniques able to gather information about sub-method elements suffer from a huge overhead. As support for the reification of sub-method elements is crucial for our work, we seek for approaches capable of efficiently gathering data beyond the method boundary.

Partial behavioral reflection is a technique provided by Reflex [TANT 03] for Java and by Reflectivity [DENK 07] for Smalltalk. This approach allows us to selectively reflect on specific parts of a program's execution, such as only on specific classes or particular methods, thus limiting the amount of data gathered and consequently also the performance overhead. This approach can collect data at different levels of granularity, also at a sub-method level [TANT 03]. The intention of partial behavioral reflection is to introduce a layer of abstraction between the low level details of the implementation language and the concept of capturing and reifying high level runtime events such as message sends and variable accesses [DENK 07]. Partial behavioral reflection features a convenient specification of the dynamic data to be gathered.

MAJOR [BIND 07] is an aspect weaving tool enabling comprehensive aspect weaving into every class loaded in a Java VM, including the standard Java class library, vendor specific classes, and dynamically generated classes. MAJOR is based on the standard AspectJ [KICZ 01] compiler and weaver and uses advanced bytecode instrumentation techniques to ensure portability [BIND 07]. *MAJOR* provides aspects to gather runtime information of the application under instrumentation. The collected data is used to build calling context profiles containing different dynamic metrics such as number of created objects. *MAJOR* supports the precise selection of the artifacts about which dynamic information is gathered. In contrast to Reflex [TANT 03], *MAJOR* can collect data about any class loaded into the VM, and the overhead introduced is even lower than the one of Reflex.

Besides Reflex and MAJOR, other techniques for dynamic analysis are available, but they do not meet all of our requirements.

***J.** Dufour *et al.* [DUFO 03a] present a variety of dynamic information for Java programs. They introduce a tool called *J [DUFO 03b] for metrics measurement. *J relies on the Java Virtual Machine Profiler Interface (JVMPI), [SUN 00], which is known to cause high performance overhead and which requires profiler agents to be written in native code. Other profilers based on the JVMPI or its successor, the JVM Tool Interface (JVMTI), such as JProfiler[4] or JProbe[5], also suffer from platform dependence and from limited extensibility. For this reason, these approaches are not usable for our purposes.

JFluid [DMIT 04b] (cf. Section 2.1.4) exploits dynamic bytecode instrumentation and code hotswapping to collect dynamic information. Minimizing overhead is a cornerstone of JFluid's design; it achieves this by profiling a subset of the application's methods [DMIT 04b]. By not profiling the rest of the methods the profiling overhead can be dramatically reduced. Developers select root methods of the call tree so that JFluid only instruments methods in the call subgraphs determined by these root methods [DMIT 04b], which greatly reduces the number of methods being instrumented and consequently the amount of data gathered. Additionally, JFluid allows profiling to be turned on and off at will even while the analyzed system is still running. JFluid uses a hard-coded, low-level instrumentation to collect gross time for a single code region and to build a CCT augmented with accumulated execution time for individual methods.

Sampling-based profiling techniques, which are often used for feedback-directed optimizations in dynamic compilers [ARNO 01], help to significantly reduce the overhead of dynamic data collection. However, sampling produces incomplete and possibly inaccurate information, which is not appropriate for the integration in an IDE.

Dynamic Information Visualization

Before we discuss different proposals for visualizing dynamic information, we formulate the following requirements for a visualization to be useful when embedded in an IDE:

- *Lightweight.* A visualization should not be too large to not occupy too much space in the IDE.

[4] http://www.ej-technologies.com/products/jprofiler
[5] http://www.quest.com/jprobe

- *Not overloaded.* It must not contain too much or too complex information to not further worsen the information overload in an IDE.
- *Easy to understand.* If a visualization is not easy to learn and understand, developers will not use them in their daily work.

Substantial research has been conducted on runtime information visualization. Various tools and approaches make use of dynamic (trace-based) information such as Program Explorer [LANG 95], Jinsight and its ancestors [DE P 93], or GraphTrace [KLEY 88].

Program Explorer [LANG 95] provides interactive visualizations of design patterns to better navigate and understand frameworks. It particularly addresses the scaling problem by visualizing abstract but yet accurate dynamic information which is combined with static information. However, the dynamic information is not complete, information considered as less interesting is stripped away from the visualization to make it more compact.

Jinsight [DE P 93] is a visualization tool providing several views analyzing the running of Java programs to detect performance issues. Jinsight's support to gain an understanding for the program execution is limited and its views are separated from the IDE.

Ducasse *et al.* [DUCA 04] propose polymetric views for condensed runtime information to, for instance, provide an overview of the communication between classes in the whole system. However, they analyze post-mortem data and cannot focus on specific static artifacts and their interplay. Furthermore, these visualizations are only accessible in a tool separated from the IDE, hence not interacting with static source entities [DUCA 04]. However, an integration of polymetric views showing runtime information is feasible and could be of benefit to developers during software maintenance, provided that these views specifically highlight collaboration between static artifacts.

Jive [REIS 03] visualizes the runtime activity of Java programs. This tool focuses on visually presenting runtime activity such as message sending in real time. The goal of this work is to support software development activities such as debugging and performance optimizations. *Jove* [REIS 05] is an enhancement of Jive providing more detailed dynamic information, for instance to determine the concrete code statements and instructions

currently being executed. The animations of runtime activity provided by Jive and Jove would be very difficult to integrate in an IDE as they show a vast amount of data which cannot easily be embedded in the conventional IDE perspectives. Furthermore, these two tools do not aim at supporting software maintenance and program comprehension in general, but focus more on efficiency issues of the analyzed system.

GraphTrace [KLEY 88] visualizes the behavior of object-oriented programs using graphs in which nodes represent artifacts such as objects, methods or variables while edges represent relationships between the artifacts, such as inheritance or delegation. The current activity is animated in these graphs by highlighting nodes and edges. For large systems, however, such graphs do not scale, in particular the animation of graphs is hard to perceive. GraphTrace is provided in a tool separated from the IDE and is thus not easily usable during software maintenance. Similar concerns as for Jive and Jove are raised over the usefulness of such visualizations during maintenance activities performed in IDEs.

Collaboration Browser [RICH 02] recovers object collaborations from postmortem execution traces and identifies collaboration patterns. A pattern is displayed as a UML sequence diagram in a tool separated from the IDE. Additionally, the collaboration browser allows for the querying of the recovered collaboration patterns. This approach requires detailed knowledge about the system implementation to reduce the amount of information displayed in the diagrams, which renders the approach less usable for unfamiliar systems. Furthermore, no interaction with the static view on the system is possible, *i.e.* developers cannot use this browser to maintain the system.

Shimba [SYST 01] is an environment for reverse engineering Java systems by combining static and dynamic analysis. Shimba visualizes system artifacts and their static and dynamic dependencies (inheritance, invocation, containment, etc.). Shimba generates scenario diagrams to represent execution traces. As these traces are usually large, it is often difficult to apply Shimba in specific maintenance tasks and to seamlessly integrate its visualizations in the IDE.

Gammatella [JONE 04] focuses on visualizing runtime data from deployed software systems. The analyzed systems can be represented at three different levels: statement, file, and system level. At the statement level, a statement is colored if it was executed. At the file level, each

source line is colored in a miniaturized view as a horizontal line of pixels, as in Microprints [DUCA 05a] or Seesoft [EICK 92]. At the system level, Gammatella uses a treemap visualization in which each node represents a source file and its size the number of executable statements in this file. At all levels, Gammatella chooses the colors to catch the attention of developers for source statements or elements that are, for instance, slow to execute. Gammatella is a pure analysis tool, particularly suited for profiling deployed applications. It is of limited use for software maintenance tasks and not integrated in any IDE.

2.2.3 Summary

Concerning dynamic data gathering, the best suited technique fulfilling all our requirements is, for Smalltalk, partial behavioral reflection as provided by Reflectivity. For Java, MAJOR is more appropriate than Reflex as it can cover all Java classes, including JDK and dynamically loaded classes. All other proposals have some flaws and thus cannot be used to achieve our goal of integrating runtime information in the IDE's static code perspectives.

Although the presented techniques and visualizations to present the gathered dynamic information have their respective use cases, we believe that most of them are too heavyweight to be integrated in IDEs. As IDEs overload developers with information, we must not add complex visualizations that occupy either much screen estate or otherwise impose a cognitive burden on developers. Rather we aim at presenting the gathered dynamic data using lightweight approaches that can be seamlessly integrated in the conventional IDE tools and perspectives such as package explorer or source editor. Such visualizations include, for instance, heat maps, icons, or, to some degree, polymetric views, along with textual presentation of dynamic information.

2.3 Conclusions

We conclude this section on the state of the art in research on development environments, software analysis, and software visualization by summarizing the most important lessons learnt:

- The existing research proposals and tools enhancing or enriching IDEs are not capable of satisfactorily solving the problems of IDEs we raised, that is, information overload and narrow focus on static software structure.

- We learnt from existing work that in particular techniques such as highlighting artifacts of interest (Seesoft, Microprints, Mylyn, and others) are a useful means to visually and non-intrusively convey information helping developers to focus on important and relevant artifacts, thus mitigating the negative consequences of information overload.
- Representation of context and task-relevant information (FEAT, Mylyn) is an important augmentation of IDEs to help developers remain oriented and focused on the current software maintenance task.
- Determining collaborating artifacts based on structural information or investigation and modification activities performed by developers (FEAT, Mylyn, NavTracks, Hipikat, and others) does not yield sufficiently precise results. Thus we additionally need to exploit behavioral information to achieve precise collaboration information.
- Approaches presenting dynamic information from one single execution or just from a slice of it (Compass, Whyline, other debuggers or profilers) cannot give a comprehensive and sufficiently general view on dynamic collaborations or execution flows in hard to understand static source code. In particular for feature analysis, a permanently accessible representation and visualization of features is helpful to developers while maintaining software systems. Debuggers, however, usually provide volatile information not accessible from within the static source perspectives of IDEs.
- From studying several works on static analysis, we are optimistic that easy-to-understand, lightweight visualizations such as polymetric views could be appropriate means to improve the overview in IDEs while at the same time not overloading the developer with even more information. Key is that any additional means added to the IDE is well integrated with the existing perspectives and tools. Although promising, we do not follow the path of providing complex and heavy-weight visualizations such as CodeMap or CodeCity to developers in IDEs as such means might further increase the information overload.
- Our brief overview of existing work in the area of dynamic analysis revealed that only a few dynamic data gathering approaches are able to meet all our requirements formulated to obtain behavioral information required to successfully address the narrow focus of IDEs on static views on software. Partial behavioral reflection (Reflectivity) and aspect-based data gathering (MAJOR) provide appropriate capabilities.

- Existing visualizations of dynamic data are usually provided in analysis tools separated from the IDE. Their integration in IDEs is, though possible, not attractive as they usually show too much information to be of use to developers while maintaining software systems. We opt for the integration of dynamic information into the existing IDE tools locally to the views on static source elements and aim at using lightweight visualizations such as heat maps or polymetric views.

Part I

Mitigating Information Overload in IDEs

The first part of this dissertation introduces proposals aiming at alleviating the information overload in IDEs. The work presented in this part later on allows us to integrate dynamic information into the purely static source perspectives of modern IDEs.

We discuss three distinct proposals in this first part of our work:

- *HeatMaps* (Chapter 3) highlight relevant source artifacts in an IDE's source perspectives to be able to more efficiently identify important and interesting elements of a software system.
- *SmartGroups* (Chapter 4) provide workings sets of task-relevant source artifacts to enable developers to focus on a small fraction of the entire source space and thus reduce the negative impact of information overload.
- *AutumnLeaves* (Chapter 5) performs "housekeeping services" by automatically removing from a developer's workspace unused windows or tabs and thus tackles the plethora of open windows with which a developer is overloaded in an IDE.

We conclude the first part of this dissertation by critically discussing these three proposals to reveal to which degree they solve the shortcomings of IDEs related to information overload.

Chapter 3

HeatMaps – A Navigational Aid

3.1 Introduction

3.1.1 Positioning *HeatMaps*

In this chapter, we present *HeatMaps*, an approach we implemented in the Squeak and Pharo Smalltalk IDE to address the problem of being overloaded with information in the IDE. In particular when working on a large software system containing many hundreds or even thousands of classes, developers ultimately have problems gaining or maintaining an overview of this system, in particular if they are not familiar with it. *HeatMaps* tackle this problem by highlighting in the IDE views the artifacts of interest with a heat color; the more red an artifact appears the more important it is considered to be for the task-at-hand. Thus, *HeatMaps* reduce the number of artifacts developers have to deal with by allowing them to focus on artifacts colored in a "hot" color. Hence, developers can more quickly gain an overview of the system. Additionally, *HeatMaps* also provide some sort of context as it supports multiple means to determine the importance of artifacts. Depending on the current development task, the developer selects a particular HeatMap to be displayed. The entities colored in this HeatMap are likely to be relevant for the current working context.

Figure 3.1 lists the IDE problems the *HeatMaps* approach addresses. The figure shows that *HeatMaps* are helpful during all development ac-

	Problem \ Activity	Feature investigation	Feature implement-ation	Artifact investigation	Dependency investigation	Runtime interaction investigation	Artifact usage investigation	Execution pattern investigation	Quality assessment	Domain concept analysis
Overloaded views	Information overload	✔	✔	✔		✔				✔
	No overview	✔	✔		✔				✔	
	No context, task support	✔	✔							
Static views	Distributed artifacts	✔		✔	✔	✔		✔	✔	✔
	Collaboration hidden	✔		✔	✔	✔		✔	✔	✔
	Execution paths hidden			✔	✔	✔	✔	✔		

Figure 3.1: *HeatMaps* highlight relevant artifacts to reduce the information overload and increase the overview in static source views. *HeatMaps* also provide limited support for the representation of context and helps developers to identify distributed artifacts that are conceptually related. As *HeatMaps* can also take into account dynamic information, they make execution paths more tangible by highlighting executed artifacts.

tivities identified in Section 1.1.1. In the remainder of this chapter, we motivate in detail the need for *HeatMaps*, discuss the various kinds of maps we provide and the exact problems they tackle, report on how we acquire the information for the HeatMaps, and ultimately validate this work.

3.1.2 Introduction to *HeatMaps*

Conventional IDEs enable the exploration of a software system principally by providing views and mechanisms based on the static structure of the source code. Object-oriented language characteristics such as inheritance and polymorphism can lead to conceptually related code being scattered over many different source artifacts [DUNS 00, WILD 92]. This can lead to an unfocused, undirected navigation of the source space, resulting in the same entities being browsed several times during the same working session. Empirical experiments have shown that during a one day coding session, developers browsed 95% of all visited methods more than once [PARN 06].

IDEs offer little support to efficiently navigate the source space aside from the static system structure. Information about previous navigation, about the system's dynamics or its evolution, is not exploited. Previous research efforts such as NavTracks [SING 05] and Mylar [KERS 05] show that this additional information provides useful insights to a developer exploring a system or relocating previously browsed entities. If a developer has for instance access to historical information about her own navigation

or that of other developers, she will be able to locate previously navigated entities more quickly [SING 05, KERS 05].

Although gathering additional information about navigation, history, or even the dynamics of a system may not be particularly challenging in itself, representing and displaying the vast amount of information gathered in the constrained IDE space without further increasing the information overload is inherently difficult. In this chapter we tackle the following research question: "*Is there a unifying mechanism to represent the complex information that developers face in the context of a constrained IDE space while working on a development task?*" This question is then further divided into the following issues:

- How can a uniform mechanism represent various kinds of complex information in an IDE?
- What different kinds of information are of use to a developer while performing various tasks on a large software system?

In this chapter we introduce a simple and uniform mechanism, called *HeatMaps*, to represent complex information in an easily understandable way in any IDE. This mechanism allows us to seamlessly integrate such information without considerably increasing and worsening the information overload in the IDE perspectives. Instead developers subjectively get the impression of dealing with less information as *HeatMaps* help them to quickly identify the relevant information.

A HeatMap maps all source artifacts presented in the IDE to colors ranging from red ("hot") to blue ("cold"). Hot entities contribute heavily to a given property while cold ones contribute little or nothing. HeatMaps represent a simple and uniform mechanism as we can apply them to very different properties of software, such as how recently a source artifact has been navigated or modified, how many versions or authors an entity has, or even how much space is allocated to a method invocation. Different HeatMaps may be more suitable than others for a given task-at-hand, so the developer can configure the IDE dynamically to apply a selected HeatMap. A HeatMap can also be defined as a combination of existing HeatMaps, to simultaneously display different kinds of information.

In Section 3.2 we motivate the need for a uniform approach to represent various kinds of information in the IDE. In Section 3.3 we present the *HeatMaps* mechanism in detail. We assess the efficiency and accuracy of various HeatMaps for several case studies using a data set spanning 20 months of IDE navigation in Section 3.4. Section 3.5 discusses the strengths and weaknesses of this approach while Section 3.6 concludes the chapter with some remarks on future work.

3.2 Information Overflow and Overload in IDEs

As discussed in Section 1.1.2, development environments present vast amounts of information to developers, usually reflecting the static structure of the code. For example, in the Eclipse IDE[1] there are more than ten different types of projects, there is a huge icon set with more than a thousand different icons, and a large number of source code entities are distinguished, including packages, classes, interfaces, class hierarchies, methods, attributes, inner classes, and aspects. This vast range of information can easily overwhelm developers, making it difficult for them to focus on the particular entities relevant to a task, such as classes working together at runtime, or methods changing in tandem in every new version.

We therefore argue that there is a need for a configurable mechanism to ease navigation by highlighting software artifacts of special relevance to a given task.

Next we present a typical use case that could clearly benefit from such a mechanism.

3.2.1 Motivating Use Case

As developers we face the task of correcting a defect in a large, unfamiliar web application written in an object-oriented language. This defect occurs in a feature that has been previously implemented by another developer who has left the team. Due to our lack of knowledge about this system, we cannot easily identify the entities responsible for the broken feature. Our IDE has recorded the development actions performed by the developer while building this particular feature, so we can exploit this information. However, it is not clear how this historical information can be presented in the IDE to help us with this task. In addition to the historical navigation data, we also have access to the change logs, containing data about previous versions, commits, and authors. It is known that this kind of information can also be useful to direct the developers to software artifacts likely to contain defects [Gî 04, HASS 04, TARV 09, ZIMM 04b]. Besides correcting the mentioned defect, we are required to also boost the performance of this feature in general. We hence want to see directly in the IDE hints about the execution behavior in terms of execution time.

[1] http://www.eclipse.org

In our case we could benefit from the availability of three very different kinds of information directly in the IDE: (i) information about previous navigation and possibly modifications performed by developers in the past, (ii) information about the system's evolution, and (iii) information about the runtime behavior of the subject system.

3.2.2 Development Driven Information

There is a large range of information that is orthogonal to the static structure of a software system, but which may be of use for various development tasks. In line with Chapter 2, we consider the following kinds of non-structural information that can support developers performing the task-at-hand:

- *Exploiting navigation and modification activities.* The history of navigation and modification of source artifacts can be exploited to provide hints to developers where they may want to navigate to or what to modify in order to perform a development task [SING 05].
  - Recently browsed.
  - Recently modified.
  - Frequency of browsing.
  - Frequency of modification.
  - Modified by me, *i.e.* the degree to which an artifact has been modified by the current author, measured by number of methods or versions contributed.
  - Extent of modification, *i.e.* how many lines or methods changed in a method or class.
  - Inclusion in search results, *i.e.* how often an entity appears in the results of submitted searches.
- *Exploiting evolution history.* Change logs contain a great deal of information that can help the developer to understand how the system has evolved [Gî 04, LANZ 01, PINZ 05, ZIMM 04b].
  - Number of different authors or versions.
  - Age.light-weight
- *Exploiting execution.* Dynamic information helps developers to reason about issues occurring at runtime, such as performance bottlenecks [DE P 93, JERD 96, WALK 00, RÖTH 08a].
  - Memory consumption.

– Execution time.

Given the potential value of these very different kinds of information to help developers quickly navigate to software artifacts relevant to particular task, the challenge is to present this information in the IDE in a way which does not further overload an already complex and busy user interface. We claim *HeatMaps* to be exactly such a lightweight approach to seamlessly integrate many kinds of information into IDE perspectives.

3.3 HeatMaps

We now introduce *HeatMaps* and explain our approach in detail. In particular we explore how IDEs can use HeatMaps to display the different kinds of information seen in Section 3.2.2 with a uniform mechanism.

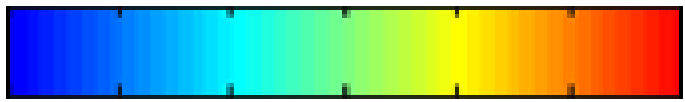

Figure 3.2: A color gradient from light blue to light red representing heat.

A HeatMap[2] employs the metaphor of heat to color artifacts: colors range from blue (cold) to red (hot) as Figure 3.2 illustrates. The "hotter" an artifact is colored, the more relevant it is meant to be for the task-at-hand. A HeatMap thus guides the developer and provides additional information about the relative importance of different source artifacts. In a large unknown system consisting of thousand of classes and methods, the *hot* artifacts are readily visible and can serve as a starting point to explore the system further. Figure 3.3 illustrates two examples where source artifacts are highlighted (i) based on the number of versions and (ii) how recently they have been browsed.

HeatMaps can be seamlessly integrated in all traditional tools of the IDE. With the help of a dedicated interface, developers choose the kind of information that the HeatMap displays, and they can also configure how different HeatMaps are combined. The HeatMap for the chosen information then appears in all views and tools in the IDE, for example, in the package browser hierarchically presenting all system entities, as well as in the hierarchy browser focusing on the class hierarchy of a selected class. Source artifacts that appear in the data history for the

[2]NB: "*HeatMaps*" (in italics) refers to the prototype tool, while "HeatMap" (unemphasized) refers to an individual map.

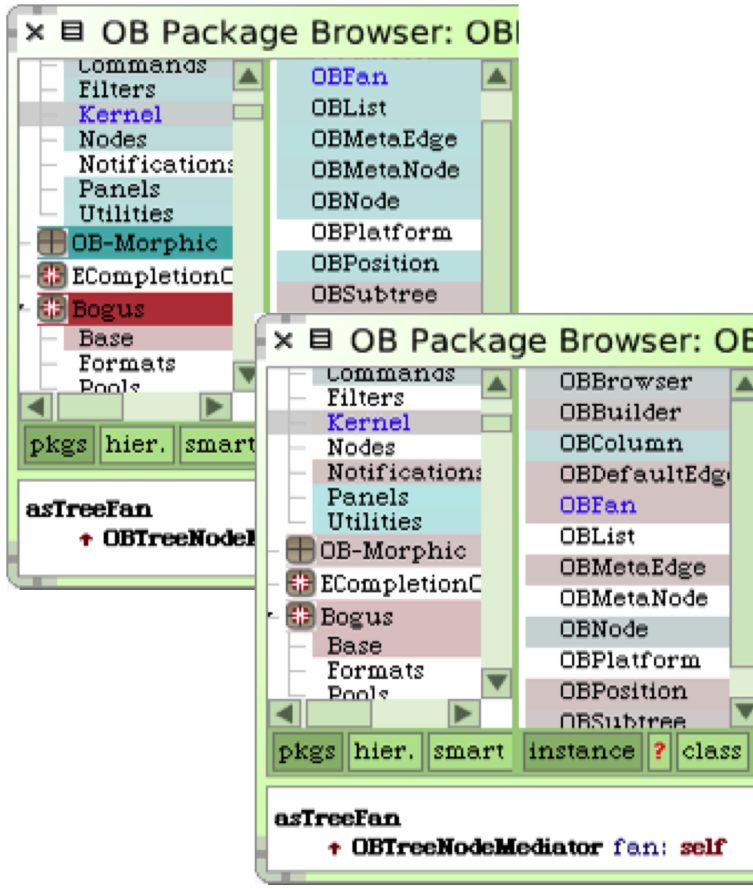

Figure 3.3: Two HeatMaps highlighting number of versions of source artifacts, top left, and recently browsed artifacts, bottom right.

selected HeatMap, such as artifacts that have been browsed or modified while correcting a defect, are assigned a background color representing their heat; artifacts not in the history are still displayed but not colored. HeatMaps do not replace or alter the display of the system's source code in any tool of the IDE, except by adding a background color to the display of source artifacts such as packages, classes, or methods. Our prototype runs in Squeak Smalltalk[3] but could easily be ported to other IDEs such as Eclipse, as the technique does not depend on any Smalltalk-specific idiom.

[3] http://squeak.org/

Typically, the navigation history, indicating how frequently entities have been browsed in the past, is a good guide to the importance of source artifacts. For a specific maintenance task other, more task-related information might lead to a better assessment of the relative importance of different artifacts. *HeatMaps* exploit all kinds of information as mentioned before in Section 3.2.2, that is, information from development activities, system evolution and execution. Developers can freely choose the appropriate HeatMap and even use maps combining different sources of information. Depending on the exact nature of the task, the system's evolutionary information might give better results than, say, information about historical navigation.

As the different HeatMaps to visualize the heat of an entity are based on very different kinds of information, we briefly describe the way in which heat is computed for two classes of HeatMaps, namely *Time-based* HeatMaps and *Metrics-based* HeatMaps.

Time-based HeatMaps. HeatMaps highlighting recently browsed or modified entities are used to reason about the time at which the navigation or modification of entities occurred. The interest in an entity usually decreases steadily after it has been navigated or edited. In Figure 3.4 we can see how with a time-based HeatMap a cold entity is associated with an early time while a hot entity is close to the current time. We assume that the interest in an entity decreases steadily as time passes by, thus an entity's color constantly "cools down". We experimented with several mechanisms to cool down an entity (cf. Section 3.4) and got best results when gradually cooling the entity as time passes by. There is a lower bound of entities' time values to take into account, determined by the size of the available history and the time passed by between now and the recorded time for an artifact. This means that if artifacts have not been covered by a relevant event for a long time, they drop out and will not be colored in this particular HeatMap. When reusing old navigation data, as in the use case described in Section 3.2.1), HeatMaps take the highest time value in the recorded data set as the current time to color the most recent items red.

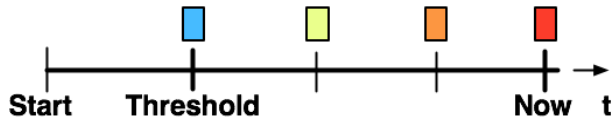

Figure 3.4: Time-based color gradient.

Metrics-based HeatMaps Frequency of browsing or modification of an artifact, and the number of developers having altered it are two examples of metrics-based HeatMaps. Such HeatMaps are used to reason about metrics associated with each artifact in the system. The higher the metric value the more important the artifact becomes. Metric values are linearly mapped to heat colors in metrics-based HeatMaps, as illustrated in Figure 3.5. To make sure that HeatMaps meaningfully highlight particularly important source artifacts, we introduce a threshold if the data set contains a wide range of different ordinal metric values. Hence we often associate *cold* not with the minimum value in the data set but with the threshold value (cf. Figure 3.5). We determine the threshold based on the system size, the size of the data set, and the distribution of the data.

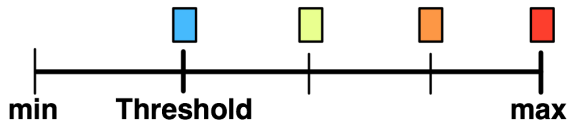

Figure 3.5: Metrics-based color gradient.

Combined HeatMaps. We assume that combining different kinds of information leads to a more accurate estimate for the source artifacts' importance than just exploiting one kind of information. Combining for example recently with frequently browsed HeatMaps is likely to better assess the developer's interest in an artifact than a single source of information. We offer two different means to combine several HeatMaps: (i) weighted linear combination of the color values of different HeatMaps and (ii) exponential decay when combining one time-based with one metrics-based HeatMap. Combining two HeatMaps linearly means that an entity once colored in blue and once in red is assigned an in-between color, if the two HeatMaps are equally weighted. It often makes sense to weight one HeatMap more than the other(s). For instance, if we combine recently browsed with recently modified, we weight the color value from the recently modified map with a weight of 2 (or even higher), as modification is rare and thus most likely increases the interest in an entity more than its navigation does. In the exponential decay combination we assume that the interest in an entity decreases exponentially over time. Obviously we are most interested in an artifact at the moment when we browse it. This event is additionally weighted with the number of times we previously browsed the same entity. From this point on, the interest in that artifact decays exponentially, similar to radioactive decay. Such a combination has the advantage that entities not having experienced

any action for a long time are still colored if they once had been very important.

How to gather the information for the HeatMaps. For many time-based HeatMaps we instrument the IDE itself to gather information about the navigation, modification, or deletion of source entities. Most metrics-based HeatMaps initially obtain their information by executing a batch process that analyzes all system artifacts to extract information such as number of versions or authors of specific artifacts. HeatMaps used to visualize behavioral information require the developer to instrument and exercise the application to gather execution time or memory usage data. For this we use partial behavioral reflection in Smalltalk [DENK 07, RÖTH 08a].

Storing, caching, updating, and exchanging the information. We store the data used by HeatMaps in a simple file format. For some HeatMaps the underlying data sets quickly grow in size, thus we cache the results of color computations. This is particularly important for aggregate entities such as packages or classes, as they aggregate the color value from their child elements (*e.g.* single methods), so rendering their color computation is more time-consuming. Usually HeatMaps are not based on an imported data set but on the data generated by the current developer in the current development session; in such cases we update the caches whenever an event occurs that is relevant to the currently selected HeatMap. These caching mechanisms make sure that HeatMaps are efficiently displayed even when their underlying data grows with the ongoing development session. The HeatMap data is easily exchangeable (*e.g.* to append it to a bug report) as it is stored in files. Thus we can easily import the navigation data generated by the developer in our use case (Section 3.2.1) to correct this defect.

3.4 Validation

HeatMaps are intended to help developers to more quickly navigate to software artifacts relevant to the task-at-hand. To be successful, HeatMaps have to fulfill at least two requirements: They need to be (i) *efficient*, so updating and displaying should not slow down the IDE, and (ii) *accurate*, that is, they should assess entities' importance properly, actually highlighting what is relevant for developers. We performed initial experiments to validate these two requirements by (i) benchmarking the efficiency of updating and rendering HeatMaps, and (ii) testing HeatMaps against an

available navigation and modification history spanning nearly two years to verify whether the various HeatMaps would haven given accurate hints to the developer. Finally, we report on an informal user experiment we conducted with developers using *HeatMaps*.

3.4.1 Efficiency of HeatMaps

We tested the performance of the *recently browsed* HeatMap by observing the time it takes to add new elements to the database, including updating all dependent HeatMaps, refreshing all involved caches and updating the visualization, and we measured the time to actually color all artifacts for a particular HeatMap in the whole system. The system we used is the Squeak Smalltalk system itself, consisting of 3180 classes and 57400 methods. We measured the display of HeatMaps in the system browser that shows all system entities. Updating the *HeatMaps* database upon navigation activities causes a non-measurable slowdown in the range of some milliseconds. Coloring the whole system with a new map affecting more than half of all entities took less than a second. Thus we consider *HeatMaps* to be an efficient means to visualize information in IDEs.

3.4.2 Accuracy of HeatMaps

In this section we evaluate the accuracy of various HeatMaps and their combinations using a benchmark.

Procedure. In a nutshell, the benchmarking procedure we implemented replays a recorded sequence of interactions, and measures the color of each element that was interacted with (in sequence) according to the HeatMap. The warmer the element is, the more accurate the map is. The sequence of interactions we replay consists of nearly 90'000 navigation and modification events recorded in an IDE while developing and maintaining a medium-sized system (consisting of 7000 methods in 700 classes) used to analyze software evolution over the course of 20 months. Benchmarks have the advantage of being easily replicable, ease the comparison of results, and can be used to test a restricted functionality, such as the effect of the weight used in the combination of different HeatMaps. The same approach has been used by other researchers to evaluate similar works such as code completion engines [ROBB 08].

We implemented two variants of the benchmark, corresponding to two distinct use cases for *HeatMaps*:

1. In the *Monitoring Use Case* the developer uses HeatMaps in her daily work. Information used in a HeatMap is continuously gathered and displayed in the IDE, so when she navigates to a new artifact, the recently browsed HeatMap immediately takes this event into account.
2. In the *Historical Use Case* the developer does not record events about her own development but imports a recorded history of another development session, for example, a session recorded by another developer while implementing a feature. This historical data is assumed to be read-only, that is, newly created events are not added to the *HeatMaps* database.

Evaluation. To simulate the first use case we create an initial database with the first 500 records in the history, test for all following elements the color value they would be assigned in a particular HeatMap, and add the tested element itself to the *HeatMaps* database. The second use case is similarly simulated; here we vary the records added from the history to the *HeatMaps* database starting at the beginning of the history with a database size of 500. We then test the 100 elements following next in the history. Afterwards we create a new database with the next 500 elements after the 100 tested elements, test the 100 subsequent elements, and so on.

Testing a single artifact means computing its color value for the currently active HeatMap, then computing the distance to red as a percentage value, so "red" is a 100% fit, "blue" and not colored a 0% fit, and values in-between are interpolated. This procedure assumes that if the developer in the history selected an artifact and a HeatMap colored it red, then the HeatMap would have successfully guided the developer to the right artifact. The percentage values are aggregated for all tested elements to form an average result for the whole HeatMap using the given history. We compute accuracy for the Monitoring Use Case as follows:

$$Accuracy = 1 - \frac{\sum_{i=d}^{n} dist(CV(x_i), RED)}{n - d + 1}$$

where d represents the size of the initial database, n is the final size of the database, x_i is the ith element, $CV(x)$ is the color value assigned to element x, and $dist(cv_i, cv_j)$ is the distance between two colors. For the Historical Use Case, the accuracy computation works similar with d equal to 500 and n equal to 600. As the window containing the analyzed hundred elements in the data set slides over the entire set, we take the mean of all accuracy values obtained for each window individually to compute the final accuracy value for the entire historical data set.

HeatMap	Accuracy
Recently browsed	74.48%
Frequently browsed	21.08%
Recently modified	34.52%
Frequently modified	4.01%
Artifacts' age	43.12%
Number of versions	$< 1\%$
Recently and frequently browsed *combined*	73.24%
Recently and frequently modified *combined*	39.17%
Recently browsed, recently modified *combined*	74.48%
Recently browsed and age *combined*	48.56%
"Best of everything"	75.91%

Table 3.1: Accuracy rates of different HeatMaps in the Monitoring Use Case.

Evaluated HeatMaps. In this experiment we test six different HeatMaps: *recently browsed*, *frequently browsed* (how often the artifact has been visited), *recently modified* (created, update, moved, renamed, or deleted), *frequently modified*, *age of artifact*, and *number of versions* (how often the artifact has been committed). Furthermore, we combine different HeatMaps to test whether combined information yields better results. We combined these maps using the weighted linear combination approach and weighted the second map with a factor of 2. As stated in Section 3.3 we can give different weights to the individual HeatMaps when combining them; in this validation each HeatMap is assigned the same weight in the combinations we tested. Finally, we did a *best of everything* experiment, that is, we computed for each tested artifact the maximum accuracy achieved under all tested HeatMaps. This final experiment thus leads to the maximum accuracy rate we possibly obtain with our approach and this data set.

Table 3.1 (Monitoring Use Case) and Table 3.2 (Historical Use Case) show the various accuracy rates for different HeatMaps we obtained using the recorded developer activities.

Discussion of the results. From these results we conclude that *HeatMaps* perform similarly well for both use cases, that is, when continuously used in a development session, or when imported from a recorded history and used without taking into account events generated thereafter. The *recently browsed* HeatMap is the best performing single metric, which comes as no surprise since the past navigation actions are most likely to be a good basis to predict future navigation actions; thus our motivating use case (Section 3.2.1) should be easier to support by a HeatMap that gives

HeatMap	Accuracy
Recently browsed	68.27%
Frequently browsed	18.14%
Recently modified	39.02%
Frequently modified	3.62%
Artifacts' age	21.93%
Number of versions	$< 1\%$
Recently and frequently browsed *combined*	63.81%
Recently and frequently modified *combined*	39.02%
Recently browsed, recently modified *combined*	65.48%
Recently browsed and age *combined*	37.41%
"Best of everything"	70.36%

Table 3.2: Accuracy rates of different HeatMaps in the Historical Use Case.

us hints about what the developer browsed while originally developing the broken feature.

Modification actions lead to significantly less accurate results compared to navigation actions, as do frequency-based HeatMaps compared to recency-based HeatMaps. This is intriguing as other researchers reported higher accuracy rates for models based on modification activities [KERS 05, SING 05]. We explain our contradicting results by the fact that the used data set contains much fewer modification than navigation activities (84000 navigation events compared to 4000 modification events); thus many browsed entities have never been modified, which means that those entities are not colored by modification-based maps. We performed another experiment which tests modification-based maps only with those entities that indeed have been modified. In this experiment we obtain accuracies of 67.49% for recently modified and 31.08% for frequently modified. The low accuracy for the number of versions map is explained by the fact that just a very small percentage of methods contains more than one version. For systems with more evolutionary information available we expect much better results for maps based on such data.

To assess the fitness of this experiment we also constantly studied the ratio of entities colored by the evaluated HeatMap and all system entities. This ratio varied between 5% and 38% throughout all experiments with an average at 17%, hence colored entities clearly stand out.

Threats to validity. There are several threats to validity in the experiment we performed. Firstly, the data set we used contains all navigations and modifications occurring in one single application, thus we cannot generalize our results to other systems as well (threat to external validity). However, we consider this data set as being fairly typically for other

applications of similar (medium) size. The system experienced several extensions, changes, and refactorings. It also contained several defects that had to be addressed, thus the recorded development history covers all the typical tasks we want to support with *HeatMaps*.

Secondly, the observed navigation and modification patterns have not necessarily been effective or even optimal, for instance, if developers didn't navigate directly to the right artifacts. Since HeatMaps should guide developers effectively to the entities they have to understand or modify in the context of a specific task, the HeatMaps should make close to optimal suggestions. The recorded data set most likely does not represent an optimal navigation in all cases, thus it is likely that HeatMaps performing well in the simulated study are not necessarily optimal for the task-at-hand (threat to external validity). However, as we know that the developers generating this data set have been involved in the system's development from the start and have thus been very familiar with it, we assume that their navigation patterns are generally very directed to what they were actually looking for. The results of the experiment would have been different if we had assumed that not the navigation but the actual modifications performed indicate an optimal pattern. In that case an optimal navigation directly opens the entities to modify in order, for instance, to correct a defect. Under this assumption, the recently and frequently modified maps give much better results, namely 72.25% and 47.81%, respectively. Neither assuming that the navigation nor the modification patterns are optimal in the available data set, is fully correct. We opted for the former assumption because navigation activities are, of course, much more frequent while working in an IDE [PARN 06] than modification activities. The reliability of test results is usually higher when based on larger data sets.

Thirdly, in this experiment we did not yet distinguish between different tasks. We are going to analyze the performance of *HeatMaps* with respect to the task-at-hand in the subsequent experiment. We performed this experiment under the assumption that the recorded data represents one large task during which developers navigated optimally (threat to construct validity). A separation by different development sessions, however, would make sure that a history of one session, for example, in which a bug was fixed, does not influence the suggestions for navigation in a completely different session dedicated, say, to refactoring. However, this implicit knowledge would have rather increased the accuracy, as using only the history of a similar session to generate the HeatMaps is very likely to give better results.

Task-dependent HeatMaps. The data set with which we performed this validation also contains information about the nature of the task that has been performed at the moment in which the navigation data has been recorded. In another experiment, we use this information to compare the performance of different HeatMaps for different specific tasks, to reveal whether some HeatMaps are better suited for one kind of development task than for another. We extracted four types of major development tasks from the same data set as before, considering all 90'000 navigation and modification activities: *defect correction*, *new feature implementation*, *refactoring*, and *navigation tasks* (tasks which do not change the system, probably performed purely to gain understanding).

To identify these four types of tasks in the data set, that is, determining the set of activities representing a particular type of task, we use a semi-automatic approach: The data set itself is split into development sessions, that is, sets of activities separated from subsequent sets with a timespan of at least two hours (to not include meetings, phone calls, or lunch breaks). We assume that a task does not last longer than one development session. We manually analyze each development session to reveal whether it consists of more than one task. We assume that navigation tasks do not contain any modification activities, defect correction tasks comprise only a few modification actions, while both refactoring and new feature implementation tasks contain many modification actions. Thus, we can assume that a new type of task starts as soon as the modification patterns change. By manually inspecting what has been navigated and modified, we can more precisely identify the moment in time when the developer switched task.

To determine the accuracy of each map for a particular type of task, we use a similar procedure as in the Monitoring Use Case: The database from which the accuracy value for a particular entity is computed basically consists of all activities that occurred before the current item in the data set. The aggregated accuracy value for a particular type of task is the average of all accuracy values computed for each data set item being part of this type of task.

In Table 3.3 we report how often a particular HeatMap most accurately directed the developer to the desired entities. For refactoring and navigation tasks, the recently browsed map performs best. For defect correction and feature implementation tasks the recently browsed combined with the recently modified map performs best. We attribute this to the fact that bug correcting activities often occur after a system has been modified, thus the recently browsed combined with the recently modified map gives best results. Feature implementation tasks often occur in sequences, thus leading to the same effect as bug correction tasks. Refactoring and

in particular navigation tasks often occur after navigation activities in which developers have spotted issues or interesting code segments to be investigated further. Hence visualizing previous navigation efforts helps developers to find the entities to refactor or analyze in more detail. The results in Table 3.3 serve as a guideline: when working on a task in one of these four areas, developers obtain best results when using the suggested HeatMap. We make use of this knowledge in *HeatMaps* to suggest well-performing HeatMaps to the developer based on the task-at-hand (cf. Section 3.5). We did not test how the HeatMaps visualizing dynamic information would have performed as there is no recorded runtime data about this system available. We expect such maps to outperform others for specific bug corrections.

HeatMap	Defect	Feature	Refactor.	Navig.
Recently browsed	49.48%	50.90%	**64.27%**	**75.19%**
Frequently browsed	19.07%	20.28%	22.99%	24.82%
Recently modified	45.20%	31.73%	38.03%	28.39%
Frequently mod.	32.98%	9.64%	17.62%	11.88%
Rec. brow. & rec. mod.	**54.31%**	**51.14%**	63.00%	72.04%
Freq. brow. & freq. mod.	32.78%	44.01%	29.22%	61.76%

Table 3.3: Performance of different HeatMaps in specific tasks.

3.4.3 User feedback

In addition to the benchmark validation we also gathered feedback from developers using *HeatMaps* in practice. Four developers used *HeatMaps* over a period ranging from several hours to a week while performing various kinds of tasks such as maintaining a familiar system. We also asked one developer to gain an initial understanding for a unfamiliar system we had developed; we provided him with HeatMaps visualizing our navigation history in this system. The developers using *HeatMaps* generally appreciated their presence during their work. They considered this navigational aid to be useful; in particular they liked that fact that HeatMaps are easy to understand and that the maps apply to a wide range of different kinds of information. The colors we chose as the background for the source artifacts are considered to be non-intrusive (we opted to use a color gradient from light blue to light red to obtain soft colors). All participants stressed the importance of suggesting task-dependent HeatMaps; although the IDE should suggest, based on the developer's characterization of the task-at-hand, the best suited HeatMap, the engineers still want to be able to customize the automatic suggestion.

After using *HeatMaps* for a while, one developer considered the *frequency* and *recently browsed* HeatMaps to be most useful when he was

interested in understanding the system in general; for addressing a specific maintenance task, he opted for HeatMaps focusing more on that task, such as HeatMaps showing evolutionary information about a specific part of the system or information involving frequency of modification, or the number of versions, not just navigation.

These early user comments offer a promising feedback about how useful *HeatMaps* can be in practice; performing a full-fledged controlled experiment we leave as future work.

3.5 Related Work and Discussion

3.5.1 Related Work

Other works pursue a similar goal, most notably Seesoft [EICK 92], FEAT [ROBI 03a], NavTracks [SING 05] and Mylar [KERS 05]. For a thorough treatment of these approaches we refer to Section 2.1. We compare these four approaches to *HeatMaps* and particularly stress their differences and limitations.

Seesoft. While Seesoft visualizes single lines of code, *HeatMaps* focus on entire source artifacts, the lowest level of granularity being the method. Seesoft's approach does not provide an overview of a system and makes it hard to identify interesting artifacts. Even for small-sized systems, Seesoft's visualizations of single lines of code do not scale and cannot contribute to a better system overview in the IDE. However, *HeatMaps* complement the approach of Seesoft well: With *HeatMaps*, developers can obtain an overview of the system and identify artifacts of interest. To explore the implementation of an artifact, *e.g.* a method, Seesoft can visualize the simple, basic metrics of *HeatMaps* (for instance, recency or extent of modification) on a per line of code basis. This is for instance interesting to reveal which lines of code have been added or modified together in the same commit or by the same author. Profiling information exploited by Seesoft also allows developers to, for instance, quickly spot lines of code with high execution frequencies. Such detailed and fine-grained information on an intra-procedural level is not provided by *HeatMaps*.

FEAT. *HeatMaps* pursue a different goal than identifying concerns as FEAT does. A HeatMap can also represent a concern though, for instance all entities colored in the same hot color can be perceived as a concern. However, the *HeatMaps* approach does not claim to identify concerns correctly, but to give hints about the relative importance of particular

artifacts for the task at hand. Entities identified as being important might or might not belong to the same concern. While FEAT aims at identifying and presenting related artifacts for a given concern to the developer, *HeatMaps* provide an overview of a system by drawing the attention of the developer to particular entities likely to be of interest. To determine this importance, *HeatMaps* take into account different data sources. *HeatMaps* allow the developer to choose the appropriate HeatMap dependent on the task and thus acknowledges the fact that there is no single answer to the question which entities need to be examined to, for instance, comprehend a system or a particular concern thereof.

NavTracks. *HeatMaps*, while using similar data as NavTracks (that is, recency and frequency of navigation) pursue a different goal, namely providing an overview of a system dependent on the development task by highlighting entities of importance. Developers can freely choose which kind of data assesses the artifacts' importance best for their task at hand. As NavTracks does not aim to provide an overview of the system to developers, the two approaches are not directly comparable. However, often all entities colored in red with *HeatMaps* are related to each other and would thus be in the recommendation lists for each other in the NavTracks approach.

Mylyn. As Mylyn, *HeatMaps* also apply a heat-based coloring scheme to highlight important artifacts, but the importance is assessed differently. While the degree-of-interest model is fixed in Mylyn, the developer can choose between different models in the *HeatMaps* approach and can even combine various models with each other to obtain better results. The *HeatMaps* models are also based on different information than that of Mylyn, including runtime and evolutionary information, such as how many different developers worked on a specific artifact in the past. For many tasks, information about previous navigation or modification is not sufficient to accurately determine the degree-of-interest, since dynamic or evolutionary information is likely to give better results. For example, when addressing a software regression, taking into account evolutionary information about who changed what in the system gives appropriate hints to developers about what they should browse. For this reason, *HeatMaps* provide suggestions to developers what information, including combinations thereof, gives best results for which kind of task. Considering the nature of the task when identifying important artifacts is likely to give better results than the rather strict degree-of-interest model provided by Mylyn. However, we have not yet formally compared the

two proposals to each other, mainly due to the fundamental differences in languages and IDEs (Eclipse and Java versus Smalltalk).

3.5.2 Discussion

Next we discuss several important aspects of our proposal: (i) combining or aggregating different information from single HeatMaps, (ii) task-dependent or goal-oriented usage of HeatMaps, and (iii) studying its limitations.

Information aggregation. From the validation in Section 3.4.2 we learn that combining HeatMaps does not appear to have a significant positive effect on the accuracy of a HeatMap. The accuracy of combinations heavily depends on the HeatMaps used, on their weighting, and on the data set of recorded activities. As the experiment studying the task-dependency of HeatMaps reveals, combined HeatMaps can outperform single maps for specific goals or tasks, as was the case for defect correction and implementation of new features (cf. Section 3.4.2), where a combination of the recently browsed map with the recently modified map performed best, also better than the recently browsed map alone. Instead of combining HeatMaps we could also already take into account the different actions performed by the user when initially building a single HeatMap. Mylar [SING 05] for instance creates a degree-of-interest model in which not only navigation but also each key struck during the modification of an artifact directly increases the interest value.

As one of our primary goals with this approach is to freely combine, exchange, and distribute the underlying data for *HeatMaps*, as well as to have a uniform approach to efficiently display very different information in the IDE, we deliberately keep the information used in single HeatMaps as simple as possible, even though gathering this information is often complex or time-consuming, as is the case for HeatMaps presenting dynamic information. This enables the developer to select from a wide range of information that which best fits the specific task-at-hand; the IDE supports the developer hereby by offering suggestions for proven combinations.

Task-dependency, Goal-orientation A navigational aid such as *HeatMaps* should ultimately guide the developer towards her goal, for example, the entity she actually needs to modify to correct a defect. In software maintenance the goals can be very diverse — gaining an understanding for the software is usually a prerequisite to attain any goal when

maintaining software. *HeatMaps* contribute to program comprehension by highlighting entities according to their importance. As an artifact's importance depends highly on the programmer's task and on the concrete goal she is pursuing, *HeatMaps* can visualize a wide range of information and propose suggestions what configuration or combination of different HeatMaps are most useful for a specific task. Developers can further refine the suggestions given by the IDE.

For many tasks and goals it may be obvious what information is likely to be most useful for identifying the important entities. For example, to optimize performance, a HeatMap highlighting heavy computations is a natural choice. Not only the HeatMap itself but also the nature or age of the data it visualizes influences the accuracy. From our experience we know that when starting a new task, the data for the HeatMaps should either be freshly recorded from scratch or originate from a similar task, otherwise the assessment of the entities' relevance is not accurate enough to properly guide the developer. For this reason *HeatMaps* provide the means to easily store the used data for later reuse in another, similar task (as done for the running use case in Section 3.2.1). Particularly useful is saving the *HeatMaps* data used while correcting a bug together with the bug report, thus giving other developers in the team the opportunity to view the HeatMaps of the original developer who addressed this particular bug [SING 05].

As mentioned in Section 3.4.2 we also provide best practice guidelines to suggest which HeatMaps are most useful for which kind of task. For developers correcting defects different entities may be important than for new team members trying to gain an initial understanding for a large software system. In the future, we want to perform empirical user studies with different HeatMaps with respect to how well they perform for various tasks. Of particular interest is the impact of HeatMaps on initial program understanding, such as when a new developer joins a team.

Limitations *HeatMaps* are limited in their expressiveness, since they can, by definition, only display one ordered set of values at a time. We can circumvent this limitation to some degree by combining different HeatMaps. However, it is unclear whether we could display more than one HeatMap at the same time in the IDE by, say, coloring artifacts in several colors, or whether we could use discrete colors to visualize discrete values, such as authors, but still show the degree of realization for a variable, for example, how much an author contributed to an artifact, by displaying a gradient around each discrete color. Abusing *HeatMaps* to

visualize too complex information is likely to erode their main advantage, that is, being easily understandable.

3.6 Summary of the Chapter

In this chapter we addressed the research question whether there is a uniform means to guide developers working on various development tasks through a large software space directly in the IDE without even more overloading the existing IDE perspectives. We proposed *HeatMaps*, a uniform approach to visualize various kinds of information orthogonal to the static system structure in the IDE. Evolutionary information, information about the historical navigation and modification performed in the IDE, or dynamic information about a large software system, can direct developers to software entities important for a specific goal and hence improve the overview of the system, in particular as developers do not have to care about all system artifacts but rather just about those being highlighted by *HeatMaps*.

As software developers and maintainers face very diverse tasks, *HeatMaps* offer a flexible and configurable means to visualize different kinds of information relevant to these tasks. In particular, we provide predefined configurations of HeatMaps that are, according to our evaluations, best suited to direct developers when solving problems such as navigating a system to gain an initial or deeper understanding, correcting defects, implementing new features, or refactoring a system.

While *HeatMaps* help developers to quickly identify artifacts likely to be relevant for the current task and to thus gain an overview of the system, this approach has also some drawbacks: (i) building a mental map of the task context is difficult; the entities colored in a hot color by a HeatMap might not formulate a working set as needed for a given task and the colored entities do usually not appear in a comprehensive list but are distributed over the entire source space. (ii) A HeatMap shows the importance of artifacts at a given moment in time; as this importance evolves over time, the HeatMap also evolves (for instance, by taking into account new modification events), thus such a map does not reflect a persistent working context. Finally, (iii) a HeatMap cannot easily be stored, distributed, or manually altered by the developer; each HeatMap is bound to a particular environment and a particular computation of the importance value. For this reason, the coloring of entities cannot be distributed or adapted by the developer.

We addressed these shortcomings of *HeatMaps* by working on another approach particular tailored to represent an explicit context in the IDE.

This approach is called *SmartGroups* and is introduced in detail in the next chapter.

Chapter 4

SmartGroups – Representing Context in IDEs

4.1 Introduction

4.1.1 Positioning *SmartGroups*

This chapter introduces *SmartGroups*, a categorization mechanism for source artifacts available for the Squeak and Pharo Smalltalk IDE. *SmartGroups* alleviate the difficulties of locating and navigating in a large software space the artifacts needed to accomplish a specific software maintenance task. To do this, *SmartGroups* provide automatically built working sets to help developers identifying the source artifacts likely to be important for their current task. Such working sets hold just a small portion of all artifacts defined in a system, thus *SmartGroups* reduce the amount of information developers have to deal with and hence tackles the information overload problem of IDEs. The artifacts contained in such automatically created groups of artifacts are, for instance, determined by analyzing the recent modification or navigation history of a developer, or by exploiting historical or dynamic information, similar to *HeatMaps*.

SmartGroups also allow developers to manually create working sets to persistently represent the context relevant for a particular software task such as correcting a defect. As a working set in particular contains

conceptually related entities, *SmartGroups* also address the problem of having many widely distributed artifacts in a software system by bringing them together in the same smart group.

	Activity / Problem	Feature investigation	Feature implement–ation	Artifact investigation	Dependency investigation	Runtime interaction investigation	Artifact usage investigation	Execution pattern investigation	Quality assessment	Domain concept analysis
Overloaded views	Information overload	✔	✔	✔		✔				✔
	No context, task support	✔	✔							
Static views	Distributed artifacts	✔		✔	✔	✔		✔	✔	✔
	Collaboration hidden	✔		✔	✔	✔		✔	✔	✔

Figure 4.1: *SmartGroups* primarily mitigate the problem of information overload, represent context in IDEs, and, to a limited degree, also make explicit hidden collaboration between distributed source artifacts.

Figure 4.1 comes back to the IDE problems discussed in Chapter 1 by summarizing all problems tackled by the *SmartGroups* approach, that is, information overload, no context representation, and distant but collaborating artifacts. Thus, *SmartGroups* support developers in all typical software maintenance activities except analyzing how particular artifacts are used by other elements in the system.

The rest of the chapter discusses in detail our *SmartGroups* proposal, the motivation for its realization, its concrete implementation and integration in the IDE, and ultimately its evaluation by means of a benchmark validation.

4.1.2 Introduction to *SmartGroups*

To comprehend a software system, developers typically use a development environment (IDE) to navigate the system and to locate important artifacts, for instance a method which introduced a defect. However, the navigation of a large system in an IDE is a time-consuming task as there are many source artifacts such as packages, classes, or methods that implement this system [KO 05]. Moreover, these entities are interconnected with each other at runtime in ways that are difficult to foresee while browsing the code [DUNS 00, CHU- 03]. In particular object-oriented language features such as polymorphism, abstract types, or the use of design patterns often lead to conceptually related but scattered and distributed code [DUNS 00, WILD 92]. For instance, a system might statically refer to abstract types, but at runtime concrete sub-types of these abstract types are used [DUNS 00]. As inheritance hierarchies often consist of many classes and interfaces that might be distributed in several packages, iden-

tifying the few types actually relevant for a specific task is often difficult or time-consuming [MARC 04, SOLO 86].

The IDE usually does not support well the identification of task-relevant artifacts. Instead, developers see in their development environment just the hierarchical structure of the software system under investigation, but there is no notion of a context specifically tailored to the current task. However, the interest of the developer in specific source artifacts is typically heavily dependent on the nature of the task. To gain an overview of an unfamiliar system, a developer needs to study different artifacts than when correcting a defect in a specific functionality of a well-known application. But the IDE always gives the same view on the source space to the developer, independently of the nature of the current development task.

We propose in this chapter the inclusion of the concept of working context in the IDE. A working context is a set of artifacts relevant for a particular task. To identify these relevant entities, we define types of tasks, namely defect correction, feature implementation, and general program understanding tasks. Different types of tasks have different relevant artifacts, thus *SmartGroups* adapt how it categorizes source elements based on the nature of the task. For defect correction tasks, for example, *SmartGroups* also take into account evolutionary information, that is, artifacts that were committed to the source repository in the past to correct a defect. For program understanding tasks, mainly navigation activities performed in the IDE are analyzed to suggest relevant entities. Developers manually specify the nature of the task; *SmartGroups* use this information to associate development activities with a task type and to recommend the artifacts appropriate for the specified task type.

Besides defining the task type (defect correction, feature implementation, or system understanding), we can further characterize the nature of the task by specifying which static or dynamic parts of a system are involved when carrying it out. The involved static part is defined by enumerating packages while the dynamic part is characterized by involved software features.

We implemented our proposal as an extension to the IDE called *SmartGroups* which is available for the Squeak and Pharo Smalltalk IDE. This extension represents working context by categorizing source entities in groups. These groups are "smart" in the sense that they hold source entities automatically categorized by algorithms tailored to specific task types. These algorithms use various kinds of information such as the modification and navigation activities performed in the past on the system under study, evolutionary information such as recently committed artifacts, or dynamic information such as number of invocations of a method.

The main contributions of this chapter are (i) a thorough analysis of the difficulties in navigating the software space in IDEs, (ii) an implementation of *SmartGroups* mitigating these difficulties, (iii) the evaluation of various algorithms to automatically identify the elements in the smart groups, and (iv) the validation of how correct and accurate *SmartGroups* identify task-dependent entities.

This chapter is structured as follows: Section 4.2 thoroughly studies the problem of software navigation and comprehension in IDEs while Section 4.3 reports on related approaches such as Mylyn or NavTracks. In Section 4.4, we introduce *SmartGroups*, our implementation of a smart and automatic categorization mechanism for source artifacts in IDEs, and the algorithms and their parameters *SmartGroups* use to identify task-relevant source artifacts. Section 4.5 evaluates our proposal with a benchmark validation based on a recorded set of development activities. Eventually, Section 4.6 concludes the chapter.

4.2 Software Space Navigation Issues

As we stated in Section 1.1.2 an IDE's views on a software system are typically overloaded. It is difficult to navigate a large software space in such overloaded views, in particular as the IDE does not provide any guide to the developer how to locate relevant artifacts [KERS 05]. However, software navigation is a crucial prerequisite for program comprehension [BASI 97, CORB 89], since most software systems spread their functionality over multiple source artifacts [DUNS 00, CHU- 03, SOLO 86]. Even reasonably sized systems contain several hundreds of these artifacts (classes, methods, etc.). As conceptually related code is often distributed over the entire source space, understanding for instance a particular software feature requires developers to spend considerable time and effort to navigate a feature [KO 05, MARC 04]. During this navigation, developers often lose the overview and have to start over searching for the right path to be able to comprehend a software feature [NIEL 89a].

In this section we empirically analyze the software space navigation problem developers suffer from when working on software maintenance tasks in IDEs such as Eclipse or Smalltalk. We examine several recorded development sessions to study the extent of navigation problems in the traditional Pharo and Squeak IDE and to obtain some indicators giving evidence for the existence of navigational difficulties in IDEs. Although the two IDEs we cover in our study are dedicated to the non-file based, dynamically typed Smalltalk language, we do not expect major differences in the analysis results for other IDEs such as Eclipse or NetBeans,

Indicator	Avg. of 20 sessions
Number of window switches	38.85
Number of entities revisited	35.10
Edit / navigation ratio	9.51%
Number of navigation actions until first edit	52.14
Number of navigation actions btw. two edits	19.31

Table 4.1: Five indicators highlighting navigation issues occurring in the Squeak Smalltalk IDE.

which are used for programming in statically typed file-based languages such as Java. We further analyzed these development sessions to elicit ideas for the improvement of software navigation, in particular to reveal whether having a representation of a working context could help to cure software navigation issues.

Problem indicators. As indicators for navigation difficulties we consider the number of *window switches* (changing focus from one window to another), the number of *re-visits of source artifacts* purely for reading and understanding (without modification), the *edit/navigation ratio* (ratio of edit actions compared to navigation actions), the *extent of navigation until first edit* (how many navigation actions a developer performed until modifying the first artifact), and *the average extent of navigation between two edits* (how many navigation actions occur between two subsequent modification actions). By analyzing 20 development sessions we obtained the results displayed in Table 4.1 for these five indicators. All the recorded and analyzed sessions originate from developers working for 30 minutes on software maintenance tasks (defect correction or feature adaptation) in small or medium-sized applications with up to hundred classes in Pharo and Squeak Smalltalk.

As partially introduced in Section 1.1.2, the figures resulting from this study corroborate the hypothesis that navigating the source space in Smalltalk is often difficult. Developers very frequently switch between different windows and visit many source entities several times, even during short development sessions lasting just half an hour. Locating an artifact to be modified in order to carry out a software maintenance task requires developers to spend a considerable amount of time. This is indicated by the low edit/navigation ratio (less than ten percent). All these figures demonstrate that the amount of navigation activity required to identify an artifact to be changed is large. This is in particular true at the beginning of a task when developers perform on average 52 navigation actions before they locate the first entity they want to modify. Another indication for ineffective navigation in IDEs is the average number of navigation actions performed between two subsequent modification actions;

on average developers perform 19 navigation actions until they again modify an artifact, which we consider to be a large amount of navigation between two consequent modification activities.

Study conclusions. From the numbers shown in Table 4.1, and additionally from informal interviews and discussions we had with developers, we conclude that software space navigation is an important development activity that takes a considerable amount of time as it is not well supported by conventional IDEs. We further conclude from this study that one reason for these navigational difficulties in IDEs is the fact that development environments do not represent a working context that is adapted to the current development task. Instead, IDEs always present the entire, usually huge software space to developers without providing guidance how to locate relevant information in this space.

As we revealed in Section 1.1.2 there are other reasons why developers struggle to navigate a software system in the IDE, such as a surfeit of open views, tabs or windows, the fact that dependencies between artifacts are hidden, or because features are not explicitly represented. However, representing working context in the IDE is already an improvement as such a representation is likely to drastically reduce the amount of navigation required to discover relevant entities by enabling developers to focus on a constrained portion of the software space.

In this chapter we aim at alleviating the navigation difficulties and the information overload in IDEs by categorizing source artifacts in groups to represent a context of artifacts that are relevant for the current software maintenance task. As other works described in the literature pursue a similar goal, we first study these proposals to reveal to which degree they already achieve a representation of working context in IDEs before we elaborate in the remainder of the chapter on *SmartGroups*, our implementation of a categorization mechanism for task-relevant artifacts.

4.3 Existing Approaches

Several existing proposals also aim at presenting task-relevant entities and at representing a working context in the IDE. We have introduced such proposals in Section 2.1.3. However, these related works have several limitations and shortcomings and cannot achieve our goal of representing context in the IDE. In the following, we report on these shortcomings of existing work and how we want to overcome them.

FEAT [ROBI 03a] identifies concerns from recorded program investigation activities performed in the IDE and visualizes these concerns with graphs. However, the quality of the identified concerns is heavily dependent on how organized the analyzed investigation sessions were [ROBI 03b, ROBI 07]. Disorganized investigation sessions cannot be used to identify concerns [ROBI 03b], thus FEAT's algorithms are not robust. We tackle this problem in *SmartGroups* by exploiting more than one data source to identify entities belonging to the same context or concern. This renders the *SmartGroups* approach more robust, that is, less dependent on the quality of the analyzed transcript of past investigation activities as *SmartGroups* combine several data sources to identify related entities, such as navigation and investigation but also modification activities, and even dynamic or evolutionary information. Thus, *SmartGroups* usually identify more related entities than FEAT but might be less precise, as we consider it to be more helpful for a developer to relate an artifact to a task that is actually not relevant than to not link a relevant element. The authors of FEAT report that their concerns contain just twelve different source entities on average [ROBI 03b], while a typical smart group contains twenty or more entities.

NavTracks [SING 05] recommends source entities related to the currently selected entity by analyzing how developers navigated and modified the system in the past. With *SmartGroups* we take into account more information than just recency of navigation; we also consider evolutionary data (age, versions, or authors of source artifacts) or dynamic data such as number of invocations, memory usage, or execution time. The analysis of this data yields groups of entities that form a particular context, for instance those that are relevant for a specific software feature or that are related to a specific task such as bug correction. These groups are permanently accessible and do not depend on the currently selected artifact, thus they act as a categorization of source entities. The nature of the current programming task is an important factor for the identification of relevant entities. NavTracks recommends related files independently of the task and thus ignores the relation between tasks and importance of entities.

Mylyn [KERS 05, KERS 06] exploits programmer activity to build a degree-of-interest model for the program elements in a system and highlights the elements considered interesting for the task-at-hand. *SmartGroups* are related to Mylyn in the sense that they use similar information to automatically build groups of source artifacts, namely recency and frequency of modification and navigation of source entities. However,

as mentioned before *SmartGroups* also exploit dynamic and evolutionary information.

Another difference to Mylyn is that *SmartGroups* adapt to the nature of the development task currently being performed (either defect correction, feature implementation, or system understanding). Depending on the type of task, *SmartGroups* use different algorithms to determine the elements in specific groups. While Mylyn just provides a single and fixed algorithm to identify related entities, *SmartGroups* allow developers to influence how the approach locates relevant artifacts. Developers understand their development task and the system under study usually well enough to support *SmartGroups* in the identification process by, for example, specifying the task type and packages being involved in the task, thus we do not apply such a strict model as Mylyn which computes the degree of interest value for each artifact independently of the nature of the task and the knowledge and experience of the developer. Although developers can alter the elements shown as relevant for the task in Mylyn, they cannot influence how they are initially computed.

4.4 *SmartGroups* in a Nutshell

In this section we introduce *SmartGroups*. The focus is on the automatic categorization of entities that are part of the working context, that is, relevant for the task-at-hand. Besides providing automatically populated categories, the so-called smart groups, we also support manual categories to which developers can manually add artifacts. This is useful to group artifacts that are personally considered to be interesting, such as candidate artifacts for a refactoring. *SmartGroups* also provide a third kind of category which holds the results of search queries. These three kinds of smart groups are discussed in the following. Eventually, we describe how the *SmartGroups* view is integrated in the Smalltalk IDE.

4.4.1 Automatic Smart Groups

To automatically identify source entities relevant for a particular task, *SmartGroups* exploit various kinds of data sources, namely recorded development activities performed in the IDE, evolutionary information extracted from source repositories (versions, authors, etc.), and dynamic information extracted from program execution. All available data sources are combined to reveal task-relevant relations between source artifacts. For a list of the different kinds of information extracted from these three data sources we refer to Section 3.2.2.

By specifying in the interface provided by *SmartGroups* the type of task currently being performed, the developer supports the process of automatically identifying the task-relevant source elements. This task specification is abstract and high-level: the developer can choose between defect correction, feature implementation or adaptation, and general program comprehension tasks. Note that there is no dedicated task type for refactoring tasks; usually they are considered to be defect correction tasks in *SmartGroups* (cf. Section 4.5.1). This task specification can optionally be further refined by enumerating system packages that are relevant for a task or by characterizing one or several features with which the task-at-hand is concerned.

As soon as the developer has specified the nature of the current task, *SmartGroups* analyze its various data sources based on the given task specification. Recorded development activities are analyzed with regard to the task type developers performed during the recording. We assume that the same types of tasks involve similar entities; for instance, bug correction is likely to involve certain kind of entities, such as recently added or modified elements [GRAV 00], elements that contained bugs in the past [TARV 09], or that have been frequently changed [GRAV 00]. Thus, *SmartGroups* specifically take into account recency and frequency of modification to suggest relevant entities for defect correction tasks. Typically, more artifacts are navigated than modified; thus additionally to entities frequently or recently modified we also consider entities that have been frequently navigated but not modified to be task-relevant; the importance of such entities depends on the type of task.

SmartGroups suggest either methods or classes as entities being relevant for a task. Packages for instance are not considered, mainly because they are rarely modified and their navigation is not meaningful in the sense that developers, for instance, still do not know where to look for the cause of a defect in a package consisting of many classes and methods. In object-oriented applications developers mostly modify single methods to correct defects or adapt features. For program comprehension, the understanding of methods is also crucial, thus *SmartGroups* mostly suggest methods as task-relevant entities. Classes are also suggested, in particular for program comprehension tasks. As we do not consider the addition of a method to a class as a modification of the class itself (this is not true if attributes are added), classes are rarely modified during maintenance tasks, thus they are usually not directly considered as task-relevant for defect correction or feature adaptation tasks. However, developers are encouraged to also examine other methods of a class of which *SmartGroups* identified a method as task-relevant.

Task Type Identification. If recorded development activities do not have a task type associated, *SmartGroups* try to automatically determine this task type by analyzing the recorded activities. A sequence of recorded activities usually contains several distinct development sessions. Start and end of such a development session is either marked by the termination of the IDE or after a certain period of inactivity (two hours or more). A development session might contain more than one task. For sessions containing commits to a repository, we consider the time of a commit as the end of a defect correction or feature implementation task (see below). Sessions without commits are either considered to be a single program comprehension task, or, if they contain modification activities, as a single defect correction or feature implementation task. To distinguish between defect correction and feature implementation or adaptation tasks, we analyze the extent of modification: Sequences of development activities containing just moderate and local modification (that is, involving just a few entities) are perceived as defect correction tasks; if modification involves several entities that were changed relatively extensively, we assume a feature implementation or adaptation task.

Determining task types from evolutionary data works similar. As this kind of data does not include information about entity navigation, we basically just distinguish between defect correction and feature implementation or adaptation tasks by considering the extent of modification (both in terms of number of modified entities and added, changed, or removed lines of code in specific artifacts). Defect correction tasks usually cover just a few entities while the extent of feature implementation tasks is larger. If a programmer specifies a type of task in *SmartGroups*, this information is automatically stored in the commit message, thus we can retrieve this information from evolutionary data.

Evolutionary data extends data about recorded development activities (i) by grouping modified entities into a coherent set, that is, the entities being part of the same commit, (ii) by adding date and time, author information, and a commit message to this set of modified entities, and (iii) by finalizing a batch of modification actions. From recorded modification actions it is difficult to separate intermittent modifications from those finally solving a particular task. We expect that committed entities contain final changes while recorded modification actions are often just a step towards the final modification of a particular entity in a particular task. We thus take the time of commit as the completion time of a task, at least in cases where developers did not manually specify when a task is finished using the interface of *SmartGroups*. Furthermore, commits help to refine information contained in recorded modification activities as they usually just include entities that indeed have to be modified in order to

complete a task while they should not include entities modified to, for instance, introduce logging statements to better understand a program.

The identification of task-relevant entities based on development activity and evolutionary data works as follows for the different types of tasks:

Defect correction. First, we map particular commits to recorded development sessions to mark the end of a task. The beginning of a task has been either specified by the developer or has been automatically determined as described above. As soon as the extent of the task in the recorded development activities is determined, we extract all modification actions and the involved artifacts and count how frequently each artifact was modified. The set of modified entities is firstly ordered by frequency and extent of modification and secondly compared to the set of committed entities; modified entities that have not been committed are moved to the end of the ordered list of entities. Additionally, we also incorporate entities that have been frequently navigated but never modified. Such entities are placed at the end of the list, after those not committed. Source elements that have been recently modified or frequently and recently navigated in a development session are considered to be more important and thus move up in the list.

This procedure is repeated for all defect correction tasks in the recorded set of development activities. The lists of relevant entities from all considered tasks are merged; entities from recent development sessions are prioritized and thus appear higher in the merged list.

As defects often occur in artifacts that have been recently added to the system [GRAV 00], we increase the priority of artifacts that are young (age is measured in number of commits since an artifact has been initially added to the system). We also rank artifacts higher that have been changed in many commits or that have many different authors, as we expect the likelihood to contain a defect to be higher for artifacts with these characteristics.

The ranked list is shown in the *SmartGroups* view under the label "suggestions" as illustrated in Figure 4.2. Only the first twenty elements are shown by default. Developers are presented with all elements in the list on demand. We limit the maximum number of entities in the list to 50 elements; elements placed beyond this limit are not presented. Developers can change the position of elements in the list or manually add or remove elements, but the list can never grow beyond 50 elements. This hard limit has been empirically determined to be an appropriate

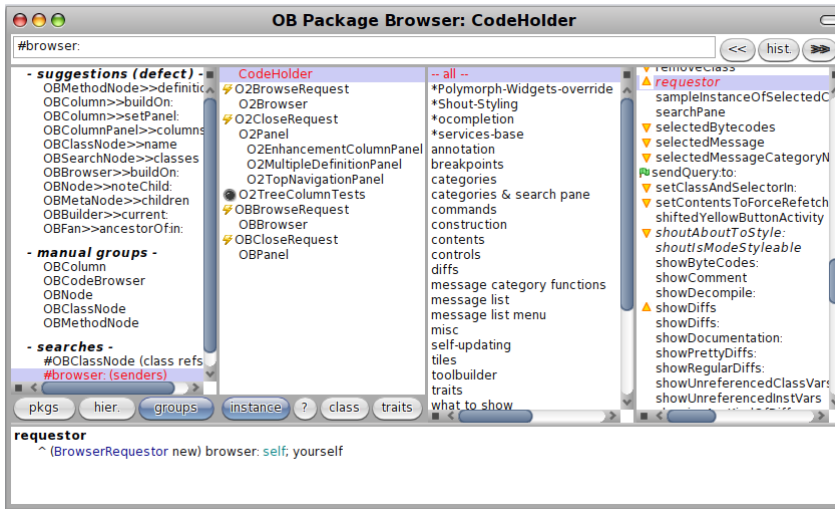

Figure 4.2: *SmartGroups* view integrated on the left side of Pharo Smalltalk's system browser, the core of the Smalltalk IDE.

compromise between having many suggestions for related artifacts and not overloading the *SmartGroups* view.

The algorithm to rank a specific artifact to determine its position in the list of related artifacts encompasses many parameters such as how frequently navigated entities move up in the list. Each element in the ranked list has an initial weight which equals its position in the list. Each parameter adds weight to some of the entities. We automatically add the maximum weight given by a parameter and increase this weight by one for entities that have not yet received weight for this parameter to move such entities towards the end of the list. Eventually, the list is sorted by the weight of entities in ascending order which leads to the final ranked list.

The different parameters in this algorithm are listed and explained in Table 4.2. We empirically determined the optimal value of each parameter by running a benchmark experiment using ten recorded development sessions. Each session contained several defect correction tasks for which we knew precisely the involved development activities. We used the recorded activities of all but one task to compute the ranked list of relevant entities for the last task in the session. We knew precisely the elements that actually had to be modified to correct the defect of this last task. We then varied in several benchmark runs the different used parameters (*e.g.* whether and how to take into account a specific parameter). Eventually,

Parameter	Description
Initialization	Initially, the list is ordered by extent of modification, that is, number of lines that are added or adapted, and by frequency of modification.
Committed entities	Entities that have been modified but not committed are appended to the end of the list in their initial order.
Frequently navigated but not modified	The 30 most frequently navigated entities are ordered by frequency and appended in this order to the end of the list.
Recent navigation	The 100 most recently navigated entities are ordered and the weight of each entity in the ranked list is increased by its rank from the 'recent navigation' list.
Frequent navigation	The 40 most frequently navigated entities are ordered and the weight of each entity in the ranked list is increased by its rank from the 'frequent navigation' list.
Recent modification	The 20 most recently modified entities are ordered and the weight of each entity in the ranked list is increased by its rank from the 'recent modification' list.
Recent dev. session	All development sessions are ordered by recency and the weight of all entities in the ranked list is increased by the rank of the development session in which they have been lastly modified.
Age (young entities ranked higher)	The 50 youngest entities are ordered by age (number of commits since creation) in ascending order and for each of these entities appearing in the ranked list we increase its weight by the rank from the 'age' list.
Number of versions	Each entity is ordered by number of versions in descending order and the weight of each entity in the ranked list is increased by its rank in the 'number of versions' list.
Number of authors	Each entity is ordered by number of authors in descending order and the weight of each entity in the ranked list is increased by its rank in the 'number of authors' list.

Table 4.2: The different parameters used in the algorithm to identify entities relevant for defect correction tasks and how they influence the order of the relevant entities.

we chose the parameters from the benchmark run which proposed a list of relevant artifacts best aligned with the set of elements that developers actually had to modify to correct the last defect in each development session. The benchmark validation principle is explained in more detail in Section 4.5.

Feature implementation and adaptation. The identification of source elements relevant for feature implementation tasks works largely in the same way as described above for defect correction tasks, except that feature implementation tasks extracted from the development activity and source code history are analyzed instead of defect correction tasks.

Parameter	Description
Initialization	Initially, the list is ordered by extent of modification, that is, number of lines that are added or adapted, and by frequency of modification.
Committed entities	Entities that have been modified but not committed are appended to the end of the list in their initial order.
Frequently navigated but not modified	The 20 most frequently navigated entities are ordered by frequency and appended in this order to the end of the list.
Recent navigation	The 200 most recently navigated and modified entities are ordered and the weight of each entity in the ranked list is increased by its rank from the 'recent navigation' list.
Frequent navigation	The 100 most frequently navigated entities are ordered and the weight of each entity in the ranked list is increased by its rank from the 'frequent navigation' list.
Recent modification	The 100 most recently modified entities are ordered and the weight of each entity in the ranked list is increased by its rank from the 'recent modification' list.
Recent dev. session	All development sessions are ordered by recency and the weight of all entities in the ranked list is increased by the rank of the development session in which they have been lastly modified.
Age (young entities ranked higher)	Not used.
Number of versions	Not used.
Number of authors	Not used.
Same author	Entities changed by the same author as the current developer are ordered by the recency of the development session in which this author changed the entity. The weight of each entity in the ranked list is increased by its rank in the 'same authors' list.

Table 4.3: The different parameters used in the algorithm to identify entities relevant for feature implementation and adaptation tasks and how they influence the order of the relevant entities.

There are also some minor differences in the identification algorithms compared to defect correction . For instance, we rank artifacts higher that have been previously modified or navigated by the same author as the current developer, as we consider it as likely that the same developer will work on similar features throughout the lifetime of a system. Thus, entities this developer changed during previous development sessions are more likely to be relevant for the current task than artifacts this developer has never touched before. We expect this effect to be less pronounced for defect correction tasks as often defects have to be urgently corrected, thus the first available developer may perform the correction and not the one who normally works on the affected feature.

Compared to defect correction tasks, we slightly adapted the parameters of the identification algorithm as depicted in Table 4.3. This adaptation was necessary because the benchmarks executed to gauge the

Parameter	Description
Recent navigation	The 100 most recently navigated entities are ordered and the weight of each entity in the ranked list is increased by its rank from the 'recent navigation' list.
Recent dev. session	All development sessions are ordered by recency and the weight of all entities in the ranked list is increased by the rank of the development session in which they have been lastly modified.
Search results	The weight of all entities to which developers have navigated from search results is not increased while the weight of all other entities is increased by ten.
Reading time	All entities are ordered by reading time in descending order and the weight of each entity in the ranked list is increased by its rank in the 'reading time' list.
Time of visibility in a view	All entities are ordered by the time they are visible in a view in descending order and the weight of each entity in the ranked list is increased by its rank in the 'time open in view' list.

Table 4.4: The different parameters used in the algorithm to identify entities relevant for program comprehension tasks and how they influence the order of the relevant entities.

parameters for feature implementation tasks yielded best results using different parameters values, since the nature of feature implementation tasks differs from defect correction tasks.

General program comprehension. Identifying source elements relevant for program comprehension tasks differs from the procedure discussed above as this type of task does not encompass any modification, thus we cannot consider evolutionary information or modification activities for the identification process. We only take into account navigation activities for program comprehension tasks. We build the list of related entities in the following way: (i) the initial list of related entities is ordered by how often an entity was navigated, (ii) recently navigated entities are ranked higher, (iii) entities which developers selected in the result lists of searches are considered to be more important, (iv) the more time developers spent reading a specific artifact, the more importance we assign to it (an entity's "reading time" is measured in the outlier-adjusted time spent between selecting this entity and selecting the next one), and (iv) the longer a view on a particular entity is open, the higher we rank this entity. Table 4.4 depicts the different parameters used in this algorithm.

For program comprehension, entities added or changed during defect correction and feature implementation could also be highly interesting. Thus, when identifying the entities relevant for a pure program comprehension task, we also consult the ranked list of the other two task types. The ten top elements appearing in the ranked lists of the two other task

Parameter	Description
Not used artifacts	All artifacts not used in the recorded execution of a feature are moved to the end of the list. Thus, such entities appear after all used entities in the order they had in the original list.
Frequency of occurrence	All used entities are ordered by frequency of occurrence in the method call tree and their weight in the ranked list is increased by the rank they have in the 'frequency of occurrence' list.
Level in tree	All used entities are ordered by the highest level in which they appear in the method call tree and their weight in the ranked list is increased by the rank they have in the 'tree level' list.

Table 4.5: The different parameters for considering dynamic information to refine the ranked list of relevant entities.

types are also taken into account for program comprehension tasks; they either move up in the ranked list of the latter if they have already been identified as relevant for the program comprehension task, or otherwise are appended to the end of the list and marked with a special annotation.

For all types of tasks, we can additionally consider specific packages, authors, or time frames, if programmers opted to specify such information to identify task-relevant source artifacts. In this case, only development activities matching the specified criteria are considered to identify task-relevant entities, for instance just artifacts of a specific package.

Inclusion of dynamic information. Behavioral information is not always available, thus we do not include such information in the basic algorithms identifying task-relevant entities. However, if dynamic information is available it can greatly improve the predictive quality of the algorithms used in *SmartGroups*. To gather dynamic information the developer has to run the software feature to be corrected, extended, or adapted. For the task of implementing a completely new feature, it can be helpful to execute an existing software feature related to the yet to be implemented feature. The execution of the system is analyzed using partial behavioral reflection [TANT 03] as integrated in the IDE with *Hermion*, which is treated in Chapter 7.

The collected dynamic information (basically a tree of method invocations) influences the ranked list of task-relevant artifacts identified based on development activity and source history information in the following ways: (i) the ranking of artifacts not used in the executed feature(s) is decreased, (ii) artifacts appearing several times in the method invocation

tree in different branches move up in the list, and (iii) artifacts appearing close to the root of the method invocation tree are considered as more important, thus they also move up in the list. The parameters used in the algorithm considering dynamic information are depicted and explained in Table 4.5.

Artifacts appearing in the gathered method invocation tree but not in the ranked list are only appended to the list if it has not yet reached the limit of 50 elements. These dynamically identified entities are added to the list in the order determined by number of occurrences in distinct branches of the call tree. Thus, dynamic information refines the ranked list already identified based on development activity and evolutionary information.

4.4.2 Manual Smart Groups

Besides automatically grouping entities using the algorithms discussed above, *SmartGroups* also allow developers to manually create groups and to associate entities with such groups. All kinds of source elements (that is, packages, classes, methods, class categories, etc.) can be associated with one or more groups. One smart group could for instance hold all classes and methods implementing a logging feature of an application and another smart group could contain the artifacts responsible for an export to PDF feature. Thus, manual smart groups make distributed source artifacts accessible under a name given by the developer, for instance "logging".

Manual smart groups are integrated in the same view as the automatic groups and are presented there under the label "manual groups" (cf. Figure 4.2).

4.4.3 Query Results as Smart Groups

SmartGroups offer a third kind of group holding results of submitted search queries. At the top of the system browser included in the Pharo or Squeak Smalltalk IDE there is a text field accepting search queries covering all static artifacts in the system, that is, all classes, methods but also statically defined class references, and senders or implementors of messages. After submitting such a query, the developer obtains the result in a smart group named after the search query. Such a smart group permanently stores search queries and makes their results easily accessible in the IDE. Whenever such a group is selected, the query is processed again, thus the search results are always accurate and up-to-date.

This kind of smart group tackles the problem that developers often submit several times the same search query, as the search results vanish some time after the first search, for instance because they closed the window holding the search results. In the empirical study introduced in Section 4.2 we found that in nearly all sessions, several search queries have been submitted more than once, for instance the same searches for senders of a message are very frequently submitted several times as the original window holding these senders was closed or could not be located again in the plethora of open windows [RÖTH 09a]. Smart groups for search queries appear below the manual smart groups under the label "searches" (cf. Figure 4.2).

4.4.4 Integration of the *SmartGroups* View

The *SmartGroups* view is tightly integrated in the Smalltalk IDE which is implemented with the OmniBrowser framework [BERG 07a]. This framework provides a system browser showing four columns for packages, classes, method categories, and methods, respectively. These columns are used to hierarchically navigate the source space, similar to the way we navigate a MacOS file system with the Finder. We embed the *SmartGroups* view in the first column, that is, the package column. Tabs allow developers to switch between the traditional view showing packages in the first column and the *SmartGroups* view showing smart groups. In the *SmartGroups* view we show automatic, manual, and 'searches' smart groups in this order (cf. Figure 4.2). Developers can collapse and expand each group. For the automatic smart groups, developers can change between the different types of tasks (defect correction, feature implementation, or program comprehension) to view the appropriate suggestions for relevant artifacts for each task type. Furthermore, we provide a lightweight interface to specify the type of the current task to be performed when starting to work on it. This specification can be further refined by defining feature(s) or package(s) involved in this task.

By integrating the *SmartGroups* view in the familiar IDE interfaces, we lower the burden for the adoption of smart groups presenting task-relevant artifacts. These groups allow developers to focus on (usually) small but relevant parts of the system, thus reducing the amount of information (that is, source artifacts, but also windows or tabs) developers have to deal with. As the *SmartGroups* view is not an additional, complicated tool but an embedded perspective in an existing and familiar environment, we believe that *SmartGroups* offer an appropriate means to successfully reduce the information overload in IDEs. Switching from the *SmartGroups* view to the conventional package view in which all system

artifacts are displayed is easily possible by activating the provided tab called 'packages' (cf. Figure 4.2).

4.5 Validation

This section validates *SmartGroups* by two means: (i) we evaluate how accurate the suggestions for task-relevant artifacts are and (ii) we report on the practicality of *SmartGroups* by presenting user feedback.

4.5.1 Correctness of *SmartGroups*

For the adoption of *SmartGroups* by developers it is crucial that the suggestions for relevant artifacts be accurate, that is, the automatically determined entities supposed to be relevant for the current task should meet the following criteria: (i) the suggested entities should indeed be task-relevant (high precision, few false positives), (ii) many of the task-relevant entities should be suggested (high recall, few false negatives), and (iii) the task-relevant entities should appear as early as possible in the ordered list of suggested entities to make sure that developers do not have to skim the entire list to find a relevant artifact.

Procedure. To evaluate precision and recall of the suggestions of *SmartGroups* for task-relevant artifacts, we conduct a benchmark validation similar to the one applied for the validation of *HeatMaps* (cf. Section 3.4.2); benchmark validations have already been used for similar purposes by other researchers such as Robbes *et al.* [ROBB 08]. We analyze a recorded sequence of development activities (navigation and modification actions performed in the Smalltalk IDE) accompanied with evolutionary information (commits, versions, authors). We automatically identify the task types as described in Section 4.4.1 because developers did not specify the task types during the development activities we recorded. In an initialization phase, we use the ten first tasks of each type appearing in the sequence of development activities to build the initial lists of recommendations for task-related artifacts. To measure the accuracy of *SmartGroups*, we compare the recommendation list for a particular task type with the set of entities that have actually been relevant for the subsequent task of this type. For example, the ten first defect correction tasks suggest relevant entities for the eleventh defect correction task in the recorded sequence of development activities and the accuracy of the suggestions for the eleventh task is measured. The first eleven tasks are then analyzed to build the lists of relevant entities for the twelfth task, the accuracy of

the suggestions for this twelfth task is measured, and so on until the end of the sequence of activities is reached.

Identification of task-relevant entities. For a particular task, we determine the entities that are actually relevant as follows: For defect correction and feature implementation or adaptation tasks, relevant entities are those that are committed to the source code repository during the execution of a task. The committed entities are often a subset of all modified entities as not every modification is eventually relevant for the completion of a task. For program comprehension tasks that usually do not contain any modifications or commits, we consider all navigated entities to be relevant. Thus, we assume the recorded navigation of developers to be optimal which might not be appropriate in all cases. This assumption is certainly a threat to validity of our experiment (see below).

Data set. The recorded data sets we analyzed in this benchmark stem from five different developers who contributed in total nearly 50'000 navigation and modifications events that were accompanied with 268 commits to a source repository. These developers worked on six different systems of medium size (consisting of between 300 and 1200 classes). The time span covered in the recorded sets for each system varies from three weeks to five months. For each system, we use the recorded sequences of development activities independently of sequences originating from other systems to evaluate the accuracy. At the end, we average the determined accuracy measured over all available sequences of activities.

Evaluation. To determine how accurate the identified task-related entities are, we compare the set of entities that have actually been related to the task (determined with recorded development activities and evolutionary information) with the suggestion list of *SmartGroups*. This list is ordered and contains a maximum of 50 elements. None of the recorded defect correction or feature implementation tasks spanned 50 elements (the number of relevant elements varied between one and 37). Actually relevant task entities should be included in the respective suggestion list for each task to achieve a recall of 100%. Some program comprehension tasks exceeded the limit of 50 elements. For these tasks, we temporarily allowed *SmartGroups* to suggest more than 50 entities, namely all elements it could identify as being task-relevant. To measure recall we thus count the number of task-relevant entities not identified by *SmartGroups* (false negatives). We can measure precision by analyzing how many entities *SmartGroups* suggested that are actually not task-relevant (counting false positives).

As the suggestion list is ordered, we can also consider the position of a correctly suggested element in this list. If the list has n elements and x elements are actually task-relevant (x < n), then *SmartGroups* achieved a fully correct identification if all x elements are contained in the first x elements of the suggestion list. Even when the suggestion list contains all x relevant elements, it makes a difference to a developer whether these x elements are, for instance, shown at the beginning or at the end of the list, as a developer might not browse the entire list but just check the first few elements. Thus, we calculate an ordering correctness value which takes into account the position of elements in the suggestion list. Therefore, we rate each of the x relevant artifacts that are not part of the first x elements in the list with a correctness value proportional to the distance of the element from the first x elements. Figure 4.3 illustrates the calculation of an elements's correctness value. If, for example, a task-relevant elements appears at position x + 2, its correctness value is $\frac{n-x-2}{n-x}$. Thus, the closer to the first x elements an element correctly identified as relevant appears, the higher its correctness value is. If an element is included in the first x elements, its correctness value is equal to 1.

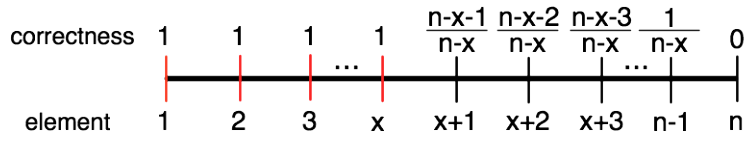

Figure 4.3: Procedure to determine the correctness of an identified task-relevant elements depending on its position.

We define the following three formulas for precision, recall, and ordering correctness:

$$\text{Precision} = \frac{\text{number of true positives}}{\text{number of true positives} + \text{false positives}}$$

$$\text{Recall} = \frac{\text{number of true positives}}{\text{number of true positives} + \text{false negatives}}$$

$$\text{Ordering correctness} = \frac{\sum_{i=1}^{x} \frac{n-x-P(i)}{n-x}}{x}$$

True positives are the relevant entities *SmartGroups* correctly identified, false positives the entities *SmartGroups* wrongly identified as being relevant, false negatives are the relevant entities *SmartGroups* could not identify. Note that it is not possible to determine the true negatives as we

Measure	Value
Number of tasks	172
Number of dev. activities	15'364
Number of commits	179
Precision	39.0%
Recall	65.3%
Ordering correctness	29.1%

Table 4.6: Results of the benchmark evaluation for defect correction tasks.

Measure	Value
Number of tasks	84
Number of dev. activities	7'982
Number of commits	86
Precision	35.2%
Recall	54.9%
Ordering correctness	27.8%

Table 4.7: Results of the benchmark evaluation for feature implementation and adaptation tasks.

do not know the exact number of source artifacts in the system at any one time during the recorded data set. Thus, we cannot compute the accuracy of *SmartGroups* defined as the proportion of true results. Precision and recall, however, give a good impression of *SmartGroups*'s accuracy. High precision and high recall values lead to a high accuracy [ZIMM 05].

n is the number of task-relevant entities identified by *SmartGroups* (that is, the sum of true and false positives), x the number of entities actually being relevant for a task. The function $P(i)$ answers the position of the correctly identified element i, that is, either 0 if element i is part of the x first elements in the ordered suggestion list or the distance between the position of i and the first x elements. Thus, the correctness value is 1 (100%) if and only if all x relevant elements appear in the first x elements of *SmartGroups*'s recommendation list.

These measures are computed for each task individually and are averaged over different tasks by computing the arithmetic mean value.

Results. We show the results of the benchmarks separated by type of tasks. The result tables present precision, recall, and ordering correctness averaged over all analyzed tasks of a particular type (except the tasks used to initialize the identification procedure). Table 4.6 presents the results for defect correction tasks, Table 4.7 for feature implementation and adaptation tasks, and Table 4.8 for program comprehension tasks.

Note that we were not able to use all recorded development activities as some were identified as belonging either to a defect correction or a feature implementation task, but there was no corresponding commit in this time period, hence we could not determine a set of entities actually being relevant for such a task. We skipped such sequences of development activities. For program comprehension tasks it is not necessary to have a corresponding commit. We ignored, however, development sessions that matched the criteria for being concerned with a program comprehension

Measure	Value
Number of tasks	143
Number of dev. activities	21'354
Number of commits	0
Precision	24.9%
Recall	20.7%
Ordering correctness	19.4%

Table 4.8: Results of the benchmark evaluation for program comprehension tasks.

task but which lasted a very short amount of time, that is, a few minutes. In general, we consider the identification of program comprehension tasks as less reliable than for the other two task types, partially because this kind of task is associated to a development session when no other task type matches.

Result interpretation. The results show that precision and recall for defect correction and feature implementation tasks is fairly high. The ordering correctness, however, is relatively low which is a sign that *SmartGroups* were often not able to correctly rank the task-relevant entities, but since *SmartGroups* could propose many candidates, the entities actually relevant were also included "somewhere" in the suggestion list. For program comprehension tasks, both precision and recall are rather low. We attribute this to the fact that identifying program comprehension tasks and separating them from other kind of tasks was more difficult than for defect correction and feature implementation tasks. Furthermore, *SmartGroups* have to rely on much less information, basically just historical navigation activities, to determine artifacts related to program comprehension tasks, while for the other types of tasks, modification activities and evolutionary information can considerably improve the prediction quality of *SmartGroups*.

Threats to validity. There are several threats to validity in our experiment:

Task type identification. As mentioned above, automatically deferring the type of task from a sequence of recorded development activities is error-prone. We might have mistaken feature implementation tasks for defect correction tasks, and vice-versa. Furthermore, since separating a development session from another one is either based on a large amount of time elapsed between two activities or by terminating the IDE, the same task might actually span more than one development session. However, for program comprehension tasks we assume that they are completed

at the end of a development session while the developer actually might have continued with this task in the next session. Similarly, it could be that at the beginning of a session, the developer worked on a program comprehension task unrelated to the defect correction task following afterwards. Yet still the entire session, at least until the first commit ending the defect correction task, is considered to be a defect correction task. Moreover, it might not be accurate in all cases to consider the time of commit as the end of defect correction or feature implementation tasks, as developers might actually continue with the same tasks even after the commit. Thus, *SmartGroups* might suggest different entities as being relevant if type, start, end, and length of tasks were correctly specified.

Granularity of tasks. The three task types we propose are very high level. There are several kinds of tasks such as performance optimization or refactoring that do not match any of the three task types. In our experiment, however, such tasks would be considered to be either defect correction or feature implementation tasks. A more fine grained categorization of tasks is more realistic and is likely to also improve the accuracy of the suggestions determined by *SmartGroups*. However, it is currently not possible to automatically defer the type of a task in a more fine grained manner. Furthermore, in reality a particular task often contains several sub-tasks matching the criteria of different task types than the main task. For instance, a defect correction task usually encompasses aspects of a pure program comprehension task. *SmartGroups* do currently not take into account the different phases occurring in a particular task. We do not know whether suggesting relevant entities matching the type of sub-tasks would yield a better suggestion quality for the overall task.

Parameter determination. We determined the different parameters and their values (cf. Table 4.2, Table 4.3, Table 4.4, and Table 4.5) used in the algorithms to identify relevant entities for specific types of tasks by running a benchmark validation using ten recorded data sets. These data sets were different than those used in this validation, but partially stem from the same developers working with the same systems as we considered in the validation. The ten data sets stem from three different developers working on four different systems. Two of these three developers also contributed data sets to this validation, and two of the four systems were also covered in the validation. Thus, the determination of the parameters is based on similar development sessions as those we used to validate *SmartGroups*. Nonetheless, we do not expect that this fact imposes a considerable threat to validity as the different development activities and tasks are fairly typical for software maintenance because all of them were concerned with software systems representative in size and complexity for many industrial applications. Thus, we expect similar

validation results even if the parameters had been gauged using other sequences of development activities.

Assumption of optimal navigation. For program comprehension tasks, we specify that all entities that have been navigated in the recorded data set are task-relevant. It is likely, however, that developers did not optimally navigate the system to answer the task-relevant questions as they did not have a perfect knowledge about the system. As the developers whose activities we recorded were very familiar with the respective systems they were working on, we expect their navigation to be effective and close to optimal, even though we do not have a means to validate how optimal their navigation actually was. We determined the indicators for navigation problems discussed in Section 4.2 for the recorded development activities and revealed that number of window switches (21.63 on average) and number of entities revisited (16.79 on average) were lower than in the data sets of developers navigating unfamiliar systems. As no modification occurs in program comprehension tasks, we could not measure indicators like edit/navigation ratio.

Generalization. It is unclear how well the recorded data set of development activities and tasks are typical and representative for software maintenance. There are several variables that might impose a threat to the generalization of the experimental results, such as the extent or severity of the defects corrected during the recorded tasks, the extent of the implemented or adapted features, the software systems being worked on, the length of the development sessions or tasks, and the developers themselves. Most developers that provided us with recorded data sets are researchers from academia working on research tools. It is impossible to say whether systems and developers from industry would lead to other results when assessing the prediction quality of *SmartGroups* for entities relevant for tasks concerned with industrial software systems, even though the considered systems are fairly representative in terms of size and complexity. Further experiments need to clarify this point. We, however, do not expect the performance of *SmartGroups* to depend heavily on the nature of the system or on how developers maintain this system. The quality of the recorded data sets on which *SmartGroups* base the prediction of task-relevant entities, particularly for program comprehension tasks, is crucial though. For this reason, recorded navigation of novice developers unfamiliar with a system should not, for instance, be used to predict relevant artifacts.

Conclusions. This benchmark validation showed that the algorithms proposed by *SmartGroups* are indeed able to properly identify task-relevant entities, in particular for defect correction and feature imple-

mentation or adaptation tasks. The predictive quality for relevant entities, however, drops for program comprehension tasks, which we attribute to the lack of substantial and reliable information to suggest related entities for this type of task. Another issue of the current *SmartGroups* algorithms is that related entities often do not appear at the top of the suggestion list, thus the developer is required to navigate in the list. We plan to improve the ranking of relevant entities by experimenting with other parameters or different parameter values than those presented in Section 4.4.1.

4.5.2 User Feedback

From the discussions with developers about the concept and implementation of *SmartGroups*, we got the following feedback:

Importance of Context. Developers stressed how important a context representation in the IDE is when we showed *SmartGroups* to them. In their daily work, they are overwhelmed with information, particularly with views containing too many static source artifacts. Developers want to be able to focus on artifacts relevant for their current task. For this reason, they considered the various smart groups as very useful. They also appreciated the categorization of search results, but asked for an automatic mechanism to remove old queries from this group as old search results are unlikely to be useful anymore after a while. The developers we discussed with were not very excited about the manual smart groups. They might use them occasionally, but it is usually too much of a burden for them to manually add entities to a smart group and to maintain these groups on a regular basis. They appreciate, however, the fact that such manual groups can be used to communicate important aspects of a system by distributing smart groups containing for instance, important artifacts of a system crucial for its understanding. In general, the ability to distribute smart groups between developers was highly appreciated.

Priority of *SmartGroups* view. Developers are tired of dealing with views showing a huge software space, for instance a tree of all packages in a system. Only a small portion of the system, usually just a few entities, is actually relevant for the current development task. Hence, developers asked to see by default the *SmartGroups* view instead of the package tree which is usually shown in the first column of Smalltalk's system browser. We thus changed the *SmartGroups* view to be activated by default while developers can switch to the traditional package tree by using the tab 'packages'.

Limited number of presented entities. Developers were glad to be able to focus on a limited number of entities (not more than 50) considered to be task-relevant. They agreed with the principle of ranking the entities by assumed relevance and to show low-ranked entities less prominently, that is, in an extended list, while only the first 20 elements are shown by default. As developers experienced that sometimes the suggested elements did not include those they actually had to modify, for instance, to correct a defect, they expressed the wish to be able to access the complete list of entities considered to be relevant by *SmartGroups*, even the elements ranked after the first 50 elements. In general, developers considered the ranking mechanism as intransparent and thus wanted to see all entities identified as possibly relevant, since the automatic ranking might have wrongly put a related entity after the first 50 elements causing it to be stripped away.

We value the obtained feedback from developers and plan to adapt *SmartGroups* accordingly.

4.6 Summary of the Chapter

SmartGroups mitigate the problem of being overloaded with information in IDEs by explicitly representing context by means of working sets consisting of a small portion of all source artifacts of a particular system. In particular the automatic identification of task-relevant artifacts supports developers to quickly locate artifacts of importance for a particular defect correction or feature implementation task. Developers have to spend less time navigating the software space as *SmartGroups* provide them with a suggestion list of relevant artifacts on which they can focus. As revealed by empirically validating *SmartGroups* by means of a benchmark validation, the automatic determination of task-relevant entities performs well for tasks encompassing modification activities and commits, but is more error-prone for pure navigation tasks performed to, for instance, gain an initial understanding for an unfamiliar system. For these kinds of tasks, *SmartGroups* offer a means to manually group relevant artifacts; such manual groups could be built by senior developers to represent the artifacts novice developers should analyze to comprehend the basics of the system. During their daily work, developers are usually not willing to extensively create manual groups, but such groups can serve as an additional system documentation and can be created and maintained by the entire development team on a regular basis.

HeatMaps also address the problem of being overloaded with too much information in the IDE by highlighting important artifacts in the entire

source space and thus helping developers to find their way through the software space. However, not only are developers overwhelmed with too many source artifacts while working in an IDE, but also with many windows or tabs they need to open to actually view, browse, and modify these entities. Maintaining an overview of all these open windows that clutter a developer's workspace even during short working session is challenging. Thus, we elaborate in the next chapter on a technique we developed to reduce the number of open windows or tabs; this technique is called *AutumnLeaves*.

Chapter 5

AutumnLeaves – Reducing the Number of Open Windows

5.1 Introduction

5.1.1 Positioning *AutumnLeaves*

This chapter presents *AutumnLeaves*, a technique to automatically close unused open windows in the Squeak or Pharo Smalltalk IDE or tabs in the Eclipse Java IDE. *AutumnLeaves* mitigates the information overload by providing "housekeeping services" to reduce the number of open windows or tabs in a developer's IDE workspace. To achieve this goal, *AutumnLeaves* continuously analyzes all open windows to check whether any open window does not align anymore with the current focus of development. *AutumnLeaves* then suggests to close such a window.

Figure 5.1 puts the problems addressed by *AutumnLeaves* in context with respect to all other IDE problems raised in Chapter 1. *AutumnLeaves* primarily mitigates the problems of information overload. Having fewer open views in an IDE's workspace also eases maintaining an overview of the system or of the current development task. *AutumnLeaves* supports developers while performing feature investigation or implementation, analysis of system quality and domain concepts, and particularly also when studying dynamic dependencies and runtime interactions, as dur-

	Problem \ Activity	Feature investigation	Feature implement-ation	Artifact investigation	Dependency investigation	Runtime interaction investigation	Artifact usage investigation	Execution pattern investigation	Quality assessment	Domain concept analysis
Overloaded views	Information overload	✔	✔	✔		✔				✔
	No overview	✔	✔		✔				✔	

Figure 5.1: *AutumnLeaves* primarily alleviates the problem of an overloaded workspace in IDEs, which, in turn, also gives developers a better overview of the system under investigation.

ing these activities developers have to open many windows to anticipate the runtime relationships between artifacts, especially when dynamic information is missing in the IDE.

The rest of the chapter explains *AutumnLeaves* in detail. After analyzing the problem, we present the concepts and algorithms behind *AutumnLeaves*, the different possible variations and adaptations of the algorithms, and finally an evaluation of the approach's accuracy by means of a benchmark validation.

5.1.2 Introduction to *AutumnLeaves*

Navigating large software systems is difficult as the various artifacts are distributed in a huge space, while the relationships between these artifacts often remain hidden and obscure [DUNS 00, WILD 92]. As a consequence, developers are forced to open views on numerous source artifacts to reveal these hidden relationships, which leads to a crowded workspace with many open windows or tabs. Developers often lose the overview in such a cluttered workspace as IDEs provide little support to get rid of unused windows. IDEs do not show how these windows are related to each other, thus developers are confronted with an immense number of independent, apparently unrelated windows or tabs to reason about. It is unclear which windows are still important and which ones have been opened to explore a branch of the navigation space not leading to the final goal. Thus developers are usually uncertain when a window will not be used anymore and are thus not willing to take the risk of closing windows potentially needed in the future. As a result, the number of open windows steadily grows.

Having many windows open at a given point in time worsens the information overload and negatively impacts system overview and navigation efficiency as developers have to spend more time locating the window of interest and as they need to keep a larger, more complex mental map of the content and purpose of each open window. Thus it

is desirable to have a minimal set of open windows at any point in time, which is likely to reduce time to navigate and maintain the working set of artifacts. It is, however, challenging to determine this minimal set, that is, the windows containing relevant, important content useful for the current problem to be solved by the developer.

As navigation is an important prerequisite to program comprehension, improving source space navigation in the IDE is an important step to better understand and reverse engineer applications while they are being developed and maintained. Literature reports that developers spend up to 35% of their time navigating software [KO 05] and up to 60% is spent with program comprehension activities in general [BASI 97, CORB 89].

To achieve the goal of reducing the number of open windows *AutumnLeaves* determines the likelihood of a window's content to be of use to the developer by relating it to all other opened artifacts. If for instance a window contains a class, a window showing a related class (such as a super- or subclass) or a method of this class is related to the first window. *AutumnLeaves* assigns to every open window a weight that will be increased upon every navigation action in the same or any other window that is related to the content showed in this window. This weight allows *AutumnLeaves* to identify those windows that have no references or only weak ones to the current development task performed by the developer. Windows with relatively little weight are steadily grayed out until *AutumnLeaves* closes them automatically (optionally by asking the developer for confirmation beforehand). This closing action occurs when the weight of a window compared to all other weights drops below a certain threshold. *AutumnLeaves* thus acts as a garbage collector for windows to mitigate the window plague with which developers are typically confronted in modern IDEs.

The research question addressed in this chapter is how to model hidden references between the various windows opened in a development session to be able to generally determine the importance of windows and in particular to identify futile, unused windows, similar to the way a garbage collector locates and terminates unreferenced objects. How should we model references between very different windows used in software development (code, debugger, inspector, or references windows) and how to represent importance of windows and changes in importance during a development task?

To the best of our knowledge, there is no proposal described in the literature which aims at reducing the number of open windows in IDEs. There are several proposals addressing the problem of being overloaded with information in IDEs, for instance by recommending related source elements while browsing particular artifacts (NavTracks), by highlighting

important, relevant artifacts (Seesoft, Mylyn), or by representing concerns or task-relevant source elements (FEAT, Mylyn). All these approaches, presented in detail in Section 2.1, reduce the number of source entities developers have to navigate or otherwise deal with, but not any approach minimizes the number of windows in IDEs.

This chapter addresses these questions by first reporting on the plague of too many opened windows in software development in Section 5.2. Second, we introduce *AutumnLeaves*, our proposal to model window references and to detect obsolete windows in Section 5.3. We validate *AutumnLeaves* in Section 5.4 concerning correctness and practicability by conducting a benchmark validation based on 25 recorded development sessions. We analyze these sessions to determine whether *AutumnLeaves* would have correctly closed a window or whether the developer used this window after *AutumnLeaves* would have closed it. This section also discusses differences between common window management techniques employed in IDEs. Finally, Section 5.5 concludes the chapter and reports on future work.

5.2 Problem Analysis: Window Plague in IDEs

Most software systems spread their functionality over multiple source artifacts (classes, methods). Even reasonably sized systems contain several hundreds of these artifacts. Depending on the programming language, these artifacts are contained in files (for instance in Java or C/C++) or are directly accessible as objects in languages such as Smalltalk [GOLD 84]. In any case, developers navigating these artifacts in modern IDEs such as Eclipse, a Smalltalk IDE [GOLD 84] or any other environment, usually view and navigate source entities by opening windows or tabs. Normally one window or tab only shows one single source entity at a time.

As soon as a window has been opened to view an artifact, it is unclear whether and how long this view is required to complete the development task. Thus developers are usually reluctant to close windows, instead they keep the views on the artifacts open as they fear to not be able to easily recover these views once closed. As a consequence, they open more and more windows, in particular when working on complex, object-oriented applications whose code is scattered over many different artifacts in statically distinct and disperse parts of the code base (for instance, in multiple packages).

We conducted several small empirical surveys and studies with developers either working with Java in Eclipse or with Smalltalk in Squeak. The fundamental difference between these two IDEs is that Eclipse works with

files containing Java classes while Squeak contains classes and methods as first-class entities not stored in files. Squeak thus supports the direct navigation of methods without first opening the declaring file and class therein. Eclipse also employs the concept of tabs (see Figure 5.2) while in Squeak, developers open full-fledged windows arranged on a desktop (see Figure 5.3). These windows can be moved, resized and minimized and often serve themselves as full-fledged browsers (that is, they contain the entire package tree from packages down to methods).

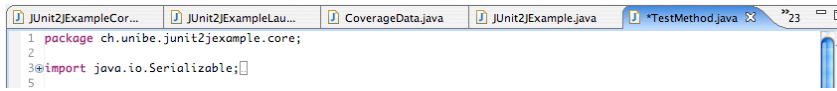

Figure 5.2: Eclipse supports tabbed browsing of the source space, but there is only space for a limited number of tabs; additional tabs are accessible in scroll list at the right.

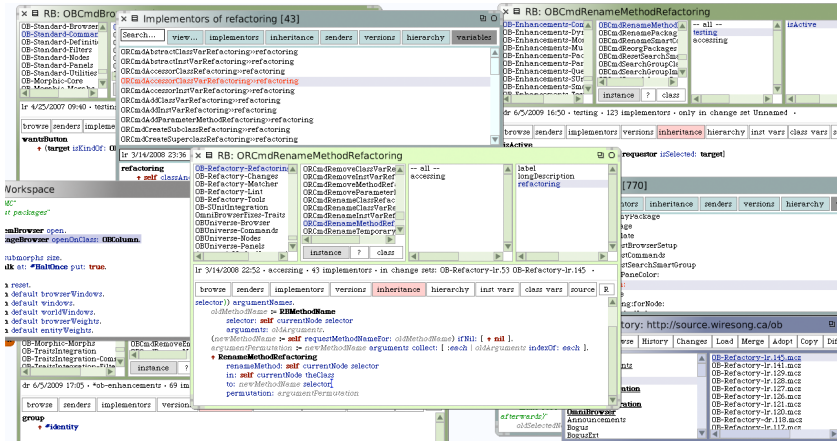

Figure 5.3: Squeak Smalltalk provides a desktop on which full-fledged windows are opened, similar as in MacOS X.

In our empirical studies we analyzed typical development sessions of developers working on smaller projects (applications with up to 100 classes of either Java or Smalltalk code). As already indicated in Section 1.1.2, we recorded the number of opened windows in total, the average number of open windows (measured in intervals of five minutes), the number of windows closed, and the number of windows opened, browsed and closed just afterwards (without changing focus to another window or tab). Additionally, we recorded the number of times people switched from one window to another and how often they visited a

Metric	Eclipse	Squeak
Number of windows opened	35.84	25.74
Avg. number of open windows	16.68	14.29
Number of windows closed	10.35	12.96
Number of windows opened and closed shortly thereafter	2.24	4.15
Number of window switches	58.90	38.85
Number of entities revisited	41.64	35.10

Table 5.1: Characteristic of the window plague in the Eclipse and Smalltalk IDE

previously browsed entity again without editing this entity on re-visit, that is, to just read and understand it again. The development sessions recorded lasted half an hour for each developer. In total we analyzed 22 such development sessions. Table 5.1 reports on the findings of these studies.

As already discussed in Section 1.1.2 the results of this study clearly show that the set of open windows in an IDE grows over time as developers do usually not regularly close windows. Most developers close a bunch of windows at the end of a development task, suggesting that they have been negatively impacted by the surfeit of windows throughout the accomplishment of the task. We can assume that human beings are not capable of cognitively handling more than seven open windows at a time [FITT 54]; however, the average number of windows is usually much higher (cf. Table 5.1). Accordingly, developers visit many entities several times and very frequently switch between windows, which are clear indications of them being overloaded with too many windows and thus having lost the overview of the system and the development task.

As Eclipse employs the concept of tabs and does not use full-fledged browser windows as Squeak, it is in general easier to re-find windows in Eclipse as they do not overlap. However, even in Eclipse the developer usually only sees between five to ten tabs in the tab bar on the screen. To access remaining open windows it is necessary to use the list next to the tab bar (see Figure 5.2). Developers reported to us that locating a window of interest in this list is very difficult and time-consuming. Usually they opt to not use this window list, but to navigate to the appropriate source artifact in the package tree and open again a view on it; Eclipse then automatically opens the window already displaying this artifact.

A common pattern of most interviewed developers to deal with the window plague is to let the list of windows grow until they are completely done with the current task. Then developers take the time to manually close all or most windows opened during the task solving process. Very

few developers close windows they consider as not needed anymore regularly during a task. However, such a procedure leads to a constantly growing list of windows clearly hampering navigation efficiency. Developers reported spending a considerable amount of time whenever they have to re-locate an open window. Moreover, they are aware that most windows they have opened become useless over time, but they are not willing or able to manually close the windows most likely not to be needed anymore.

5.3 AutumnLeaves

AutumnLeaves is an approach to overcome the previously discussed window plague. We firstly explain the basic principles behind *AutumnLeaves* and secondly report on several design considerations and variation points.

5.3.1 AutumnLeaves in a Nutshell

The ultimate goal of *AutumnLeaves* is to identify unused windows, that is, "autumn leaves" that can fall down from the tree as they are not useful anymore. *AutumnLeaves* associates a weight to each open window to indirectly model references between windows. This weight is increased upon certain user actions. Also the entities displayed in any window have a weight. This is necessary to relate entities with windows. If for instance one window displays a class, another a method or a subclass of this class, we add in our model an implicit reference between these two windows based on the entities they show. We keep the entity weight even if windows containing such entities are closed. This enables us to re-establish references between windows when the developer again opens a view on this entity in a new window.

To identify obsolete, useless windows, the weight of each window is compared to the average weight of all open windows. If a window weight is below a certain threshold of the average weight (defined as 30%), *AutumnLeaves* suggests to close the window. This suggestion is visually displayed by graying out the window or its title bar in case of tabs. Developers can always decline the automatic closing of a window, otherwise the window is closed five user actions after falling below the threshold. Additionally, the current window weight is steadily displayed in the right corner of a window to make developers aware of candidate windows for removal.

The weights (for a complete list see 5.3) and the threshold are determined by performing a benchmark validation on recorded data sets of navigation and modification activities performed by several developers working on various development tasks. Section 5.4 reports in detail about this benchmark validation. In a nutshell, we assume that *AutumnLeaves* performs well if it does close windows not used anymore later in the recorded development session. According to that idea, we ran the benchmark on 25 recoded development sessions and ultimately selected the best performing threshold and weights. We had to trade off correctness (not closing windows used later on) against effectiveness of *AutumnLeaves* (measured with the reduction in average number of open windows) and favored correctness if the results between two weight configurations were similar. We have chosen the initial weight configurations (how much specific actions should increase weight) according to a "gut feeling" for the importance of actions and varied the concrete weight around the initially chosen weight for each action by two weight points up and down.

For the threshold we experimented with all values from 5% to 50% in 5% increments. We discovered that the effect of *AutumnLeaves*, that is, the reduction of number of windows, drops quickly when lowering the threshold while correctness remains relatively stable. However, when the threshold rises above 30%, the correctness value starts to drop fast, hence we have chosen to close a window when its weight falls below 30% of the average window weight. This threshold could be further optimized, but we consider 30% to be a reasonable value.

The weights of all windows are refreshed and checked against the threshold after each user action. As a user action we consider opening a window, typing or scrolling in a window, moving or minimizing windows. To determine entity weights, we additionally consider viewing ("opening"), creating, modifying, and deleting methods and classes. The final weight of a window is the sum of its own weight and the weight of the entity it currently displays. If the displayed entity is a single method, we also add the weight of its class to the window weight (only applicable for Smalltalk as we cannot open views on single methods in Java).

To build references between windows we mostly use the entities displayed in a window. If we modify a method, we increase the weight for this entity, but also for the containing class. We thus propagate weight according to static relationships between source artifacts: From a method to its class, from a class to its direct superclass and all direct subclasses, from an inner class to its outer class, from an interface to all implementing classes. Propagated weight is always half of the direct weight for the entity: If we add weight 10 to a method, its class gets 5 points. Table 5.2 lists the different weights for all actions on entities, Table 5.3 for window

Action	Class	Method	Propagation
Viewing ("opening")	3	3	1.5
Modifying	8	10	4
Creating	4	4	2
Removing	-	-	2

Table 5.2: Weight addition to source entities upon certain actions on the same or dependent entities. Propagation means adding weight to related entities, for instance from a method to its class or from a class to its superclass.

Action	Weight addition
Initial opening	12
Moving	1
Resizing	1
Getting focus	2
Typing in it	8
Visibility (in Squeak also fractions thereof)	1

Table 5.3: Weight addition to the a window upon certain actions on this window.

actions. With these settings we obtained best results concerning correct identification of unused windows and reduction of number of windows.

Some IDEs allow developers to hide or overlap windows with others. In Squeak for instance, windows can overlap and partially or fully hide windows behind. In Eclipse, only a limited number of tabs is visible on the screen. Older tabs are only visible in the drop-down menu to the right of the tab bar. We consider visible windows to be more important than hidden ones. We hence reward fully visible windows or tabs with an additional weight point after every user action. In Squeak, we additionally take into account the degree of visibility, that is, the portion of the window at the front on the desktop and add the visible proportion of one weight point to the window weight on every user action. The desktop management facility of Squeak allows windows to be stacked.

To make sure that the weighting mechanism also properly handles windows in which no navigation actions happen but that are just selected to view their contents, we increase the weight of a window by two points when obtaining the focus. This weight is only given when the developer looks for more than three seconds at the window to only reward windows the developer intentionally selected.

We consider all kinds of windows dealing with entire source entities, that is, class browsers (showing classes and methods), debuggers, inspec-

tors, workspaces (for code snippets), list windows (list of class references, method senders or implementors, variable references, etc.). The window has to focus on a particular entity, that is, one single class or one single method. In Eclipse we consider the method in the center of the source view as the selected method. For Eclipse views such as the package explorer or the type view we consider just the selected entity but not other visible entities close to the selected one. If the entire list shown in a list view such as the package explorer was considered, we could not easily identify relations between different windows based on displayed source artifacts as most windows would be related to each other when using the entire content of list views. Other types of windows such as simple text editors, file browsers, or XML editors are not handled by *AutumnLeaves* and will thus never be automatically closed.

5.3.2 Variations, Modifications, Adaptations

Pinning of windows. One variation point is a pinning facility for windows. A window manually pinned by the developer will never be closed by *AutumnLeaves*. It will always stay there even if its weight has dropped below the threshold. Such a feature is useful for windows serving as libraries or documentation. Developers might never type in these windows, maybe not even interact with them, but still they serve a purpose to show content of interest to developers, content that is permanently important, such as a list of constants. Thus the pinning mechanism makes sure that such reference windows can stay open. Developers are free to pin any kind of windows and as many as they want. The pinning mechanism also makes sure that the windows opened for a specific task do not get closed by *AutumnLeaves* when interrupting this task to work on something else. For instance, the pinning could be categorized, so that all windows for the same tasks can be identified by the pinning category.

Visibility of windows. In Squeak, windows can overlap other windows. The visibility of a window, that is, whether it is fully visible at the front, partially visible because of other windows covering it, or totally hidden by other windows, certainly has an influence on the importance of a window to the developer. We can assume that a fully hidden window at the end of the stack is less likely to be used by the developer than a (partially) visible window. Maybe the developer even forgot about the existence of such a hidden window. We currently account for this fact by rewarding visible windows with additional weight points on each user action, in Squeak depending on the extent of visibility. In Eclipse a tab is either fully visible or fully hidden, thus a visible window always obtains

a full reward point. However, another mechanism to take into account visibility could be to check visibility just at the moment *AutumnLeaves* actually suggests to close a particular window. We could define two thresholds at which windows should be closed: a higher boundary for hidden or partially hidden windows (*e.g.* 40%) and a lower boundary for visible windows (*e.g.* 20%). We experimented with both mechanisms and report in Section 5.4.1 on differences between these two concerning correctness.

Weighting previously selected entities. Another variation point is how viewed entities should influence the weight of a window. In particular in Squeak, developers often navigate entities directly in particular windows as most windows provide browser facilities to navigate source code (Eclipse differs here as its windows only provide local navigation facilities, for instance scrolling from one method of a class to the next). The importance of such a window not only depends on the currently displayed source artifact, but also on the recent history of therein navigated artifacts. As Squeak offers means to easily navigate the history of a browser window, similar to functionality provided by web browsers, a previously viewed class is still conveniently accessible from within this window. If this displayed class is important and many other windows refer to it, then this particular window should have a higher importance even when the developer navigates further to a particular method of this class, as the old viewed entity is still easily accessible from within this window. We thus take into account in Squeak not just the currently selected entity, but also the two artifacts navigated before this entity when computing the weight of a window. The window weight is thus the sum of the weight of the window itself and the weights of the first three entities in the window navigation history. We have chosen the number three and not more to be able to react to changes in development focus, for instance if open windows are reused for a new exploration path, previous entities should not influence the window weight for too long.

Weights. The weights we have chosen (see Table 5.2 and Table 5.3) are another, important variation point. The rationale behind the currently defined weights is to take into account the content displayed in windows, that is, the navigated source artifacts, classes and methods, to be able to relate different windows to each other. However, as a variation we can also put more emphasis on actions performed on the windows themselves, such as the time spent in a window (for typing, scrolling, or having the focus). The emphasis on the entities can be further relaxed by not propagating weight from an entity to related entities (for instance, from a

method to its declaring class). Section 5.4.1 discusses the impact of weight propagation to related entities.

5.4 Validation

In this section we validate our work in two basic directions: First, we perform a benchmark validation to study the correctness of *AutumnLeaves*, that is, whether our approach correctly identifies candidate windows to be closed. Second, we report on the practicality of *AutumnLeaves*, that is, how developers assess its usefulness in practice, when working on concrete tasks in their daily work.

5.4.1 Correctness

To evaluate the correct and desired functioning of *AutumnLeaves*, that is, identifying the appropriate candidate windows for closing, we performed a benchmark validation. A benchmark validation has the advantage of being easily replicable, it eases the comparison of results, and can be used to test a restricted functionality, such as the effect of different weights on the performance of *AutumnLeaves*. The same validation procedure has been used by other researchers to evaluate similar works such as code completion engines [ROBB 08].

Procedure. Essentially, the benchmarking procedure we implemented replays a recorded sequence of user interactions that have occurred in the IDE. After each action, we let *AutumnLeaves* compute the weight of all windows as discussed in Section 5.3. If the algorithms identify a candidate window for removal, we look forward in the recorded user actions to see whether the developer ever used this window again and if so, what kind of actions he performed in this window.

In total, we analyzed 25 recorded development sessions of eight different developers. Each development session lasted between half an hour and three hours. In these sessions, very different tasks have been performed in different software systems. The development sessions used in this evaluation are not the same as those mentioned in Section 5.2 to make the results more generalizable and less tailored to the data used to identify the problem we want to solve with *AutumnLeaves*. The sessions used for the validation are longer and more complex in terms of application and task size than those used in Section 5.2. Also the developers are different persons, except one developer who contributed different recorded sessions to this evaluation as well as to the initial identification

of the problem. Most developers are either undergraduate or graduate students who worked on various tasks in research projects. We asked developers that we personally know to install our recording tool in their IDE and to submit us recorded sessions of any kind. The recording tool we implemented instruments the IDE code to send announcements about all navigation and modification activities occurring in each window we are interested in. In this validation benchmark we iterate over all recorded data sessions to find out for each window when it has been last used. In a second iteration we evaluate after each recorded action whether *AutumnLeaves* suggests to close a window and check whether this window has been used by the developer afterwards.

The participating developers described for us what kind of tasks they performed in the respective session. From these descriptions we identified six different task categories: Implementing a new system from scratch (2 sessions), implementing a new feature for an existing application with which the developer was either familiar (3) or unfamiliar (4), fixing a defect in a system (7), optimizing a system's performance (1), and a pure navigation task to gain an initial understanding for an unfamiliar software system (8). A new feature implementing task was for instance to add a navigation history button showing all previously navigated source artifacts in the Squeak browser. One navigation task for example was concerned with determining the classes responsible for rendering arrowed lines between figures in a drawing program. Most of the 25 development sessions stem from development in Squeak, while only a few (three sessions) originate from Eclipse. The systems on which developers were working had a size of approximately one to five hundred classes, except the application that has been developed from scratch which only consisted of around 30 classes at the end. After evaluating the general performance of *AutumnLeaves* we specifically test whether this performance depends on the nature of the task being performed in a development session.

The best result for the performance of *AutumnLeaves* is certainly if the developer never again used the window *AutumnLeaves* suggested to close. Even if he used the window later on in the recorded activity log, we analyze how often the window has been used and whether it has been used to navigate or modify the same entity or a related one (for instance, method or subclass of a class, a class in the same package of a class, etc.). If the window was later on used to navigate something completely different, we rate the decision of *AutumnLeaves* as correct as the developer could also have opened an entirely new window instead of re-using an existing one. If the window has been used to work on the same artifacts or on related ones, we count the related actions performed in this window and give *AutumnLeaves* a correctness rating of the reciprocal value of the counted

user actions in this particular window. If *AutumnLeaves* for instance suggests to close a window that has been used ten times afterwards, we give this decision a correctness value of 0.1. If the window has been used just once, the decision is still considered as fully correct. However, if a window has been used more than 10 times, we rate the decision of *AutumnLeaves* as entirely wrong. To obtain the final correctness rate for *AutumnLeaves* in a particular development session, we summed up all correctness rates for all candidate windows *AutumnLeaves* suggested to close and divided this by the number of total candidates.

Results. Table 5.4 shows the correctness results we got for different development sessions. Due to space restrictions we do not show all 25 but just three selected sessions, and the total performance averaged over all 25 sessions. The five selected tasks are in this order: New feature implementation (Squeak), defect correction (Squeak), navigation (Squeak), performance optimization (Eclipse), navigation (Eclipse). This table also shows the correctness value if computed in an "all or nothing" manner: Only considering a window to be closed that is never used anymore afterwards is rated as a correct performance of *AutumnLeaves*. Thus this correctness value is the percentage of perfectly correctly identified windows to be closed. We also analyzed the data sets to identify windows that have not been suggested by *AutumnLeaves* for removal, but have not been used after a certain moment. These windows can be considered as false negatives as *AutumnLeaves* should have identified them as well. A not closed window is not considered to be a false negative if it has been used until the end of the session. Such a window needs to have been accessed in the last hundred user actions of a session to not rate it as a false negative. We also determined the average time between the last usage of a window and the moment *AutumnLeaves* was able to pinpoint a window to be closed. This measure gives evidence on how fast *AutumnLeaves* is able to detect changes in the direction the development takes, for instance if the developer explores another, unrelated branch of the source space. Furthermore, we give details about the reduction of the average number of open windows (measured in intervals of five minutes).

Discussion of the results. The results in Table 5.4 show that *AutumnLeaves* usually closed windows correctly when a few usages after the moment of closing a window just reciprocally reduce the correctness rate (see definition of the correctness value introduced above). However, the correctness value dropped significantly when only closing a window never used later on is considered to be a correct decision ("strict correctness"). Nonetheless, we can still trust the suggestions of *AutumnLeaves* as

	Session 1	Session 2	Session 3	Average
Number of opened windows	82	41	109	65.20
Correctness (with some later window usage permitted)	74.18%	51.26%	47.52%	61.61%
Correctness strict	53.33%	40.00%	46.29%	51.76%
Number of windows incorrectly closed (false positives)	7	15	29	13.50
Number of windows incorrectly not closed (false negatives)	8	4	11	6.12
Time elapsed btw. last usage and closing [minutes:seconds]	8:12	7:52	12:56	10:09
Avg. number of windows without *AutumnLeaves*	25.20	15.86	32.50	28.53
Avg. number of windows with active *AutumnLeaves*	17.84	8.41	18.88	26.03
Delta in avg. number of windows	7.36	7.45	13.62	12.50

Table 5.4: Correctness, false positives, false negatives and average number of windows improvements provided by *AutumnLeaves* of three randomly selected sessions and averaged over all 25 sessions.

those windows have not been used often after *AutumnLeaves* suggested their closing and hence cannot possibly have played a crucial role in the development session.

The average number of false negatives is pretty low (6.12). We consider this to be a very promising performance of *AutumnLeaves*, in particular when comparing with the average number of opened windows (65.20). However, it takes *AutumnLeaves* a considerable amount of time (on average more than 10 minutes) to identify a window not used in the future. This means that with the current weighting mechanism, it is difficult to react on quickly changing directions in development focus. If for instance the developer finished exploring a part of the application (*e.g.* the database layer), it takes time until this is reflected in the content displayed in the various windows. The developer has to navigate further in most windows or even manually close old windows in order to make *AutumnLeaves* aware of the new development focus. We will tackle this problem in future work.

The reduction of the number of average open windows (minus 12.50) is also a positive sign for the performance of *AutumnLeaves*. We can consider any reduction of the number of open windows to be an improvement, provided that truly obsolete, unused windows have been closed. Even though we do not have evidence on how much more efficient developers are when they are confronted with fewer windows, the automatic closing of windows provided by *AutumnLeaves* certainly helps developers

to more quickly gain an overview of their workspace and of the subject system and to hence ease the source space navigation and exploration.

Another interesting result would certainly be the navigation time, that is, whether fewer windows indeed reduce the navigation time. We have not yet evaluated enough data to obtain significant results, but early evaluations indicate that the navigation time and effort is lower with fewer windows open. In future work we address this question in more detail.

Task-dependent results. The task-dependent evaluation we performed revealed that both correctness and effectiveness (window reduction) depend on the nature of the task. We obtained the highest correctness values for new feature implementation and defect correction tasks (non-strict correctness of 67.37% averaged over all such tasks). However, for these tasks the reductions of windows was, at 9.46 windows, below the average of all 25 sessions (12.5 windows). For tasks concerned with implementing a new system, performance optimizations or pure navigation, the correctness was lower (59.86%) and the reduction rate higher (13.85 windows). We attribute these results to the fact that tasks in which developers mostly navigate a constrained part of the system require opening fewer windows than tasks involving navigation of several, possibly unrelated parts of the system. *AutumnLeaves* can more correctly but less often identify obsolete windows when the general focus is on entities that are statically strongly related. Furthermore, feature implementation and defect correction tasks encompass heavily the use of structural relationships between source artifacts (*e.g.* inheritance), thus *AutumnLeaves* can more correctly identify related windows. We leave as future work to find means for weight propagation based on non-structural information to obtain better performance for the other kind of tasks such as exploration tasks.

Variations. We tested the effect of weight propagation and different threshold mechanisms with two different experiments: i) not considering propagation of weight and ii) using two thresholds instead of just one.

In the first experiment we ran two benchmarks: one using all weights as determined in the first experiment, including propagation, and another one omitting propagation of weight. The latter experiment gives slightly lower values for both correctness and reduction of average number of windows (2.56% less correct and 5.46% less reduction of number of windows). We consider propagation of weight to related entities as important, although its effect is not huge.

Instead of rewarding on each user action windows that are fully or partially visible, we evaluated in the second experiment a variation of *AutumnLeaves* which defines two thresholds, a lower boundary for visible windows and a higher boundary for hidden windows. As the latter are more likely to not be useful anymore, we assume that they can vanish earlier. For this experiment we have chosen a lower threshold of 20% of the average window weight and an upper threshold of 40% (compared to the standard threshold of 30%). The results of this experiment averaged over all 25 sessions are the following: correctness slightly increased to 63.58% while strict correctness (no later window usage permitted) dropped to 50.94%. The delta of average number of windows increased to 14.3 windows, while false negatives and false positives did not show significant changes. We conclude that this variation did not yield remarkably better results.

Threats to validity. There are several threats to validity in the experiment we performed. Firstly, the data sets we used cover pretty short development sessions (up to three hours) and were concerned with rather simple and constrained tasks. Large industrial projects may encompass longer and more complex and open tasks (threat to external validity). However, we consider these development activity logs as being fairly typically for medium-sized applications, in particular as there were four different applications involved. We can also assume that even if the tasks have been rather small and short in our data sets, the performance of *AutumnLeaves* nonetheless scales up to larger tasks as those are likely to have similar constraints and characteristics with respect to window usage.

Secondly, the fact that a window is not explicitly used anymore in the recorded data set is not necessarily a sign that it was not important later on (threat to construct validity). The developer could have looked at the content of such a window without interacting with it. At least in Squeak it is possible to have a window in the front and read its content without ever selecting and giving it the focus. However, this is not possible in Eclipse. Although such situations might have occurred in the recorded development sessions, we assume those to be very rare. Thus they should not have a significant influence on the reported results and on the prediction quality of *AutumnLeaves*. In both environments developers might have glanced at a window just shortly to find out that it the wrong one

Thirdly, developers who gave us the data sets also reported on the task they performed therein. From their description, we assigned each task to the six different task categories. We did not manually study the data sets

or ask developers further whether they worked on just one single task without any perturbations or whether they performed some other subtasks or unrelated work in this recorded activity log. Some descriptions of tasks were ambiguous as developers performed work not unmistakably assignable to one single task (threat to internal validity). Thus the task-dependent evaluation of *AutumnLeaves* contains some pitfalls regarding accuracy of the results, as the different performance of *AutumnLeaves* in some tasks is partially also explained with perturbations in the data sets and difficulties in assigning these sets to particular tasks. We consider this effect as marginal though. Another threat concerning task-dependent evaluation is that we did not have an equal distribution of the data sets on the six tasks. For instance, there was only one single data set concerned with performance optimization but seven data sets contained a navigation task. This imposes a serious threat to construct validity.

5.4.2 Practicality

While the results of the benchmark validation elaborate on the correctness of *AutumnLeaves*, the practical usefulness of our proposal is not assessed by such validation. We thus study the practicality of *AutumnLeaves* in this section.

From the discussion with developers, we learned that a crowded workspace with many open windows seriously hampers development efficiency, no matter on which task they are working. In particular when navigating software systems to reverse-engineer them, for instance to build a mental model of a system in order to be able to extend or correct particular software features, developers suffer from too many open windows, which can ultimately lead to a lost of overview. Any solution to overcome this window plague comes as a relief, developers reported. However, it is considered to be important to have full control over the windows. Developers are not willing to accept a fully automatic closing mechanism, instead they always want to have the power of veto, for instance if *AutumnLeaves* suggests to close a window actually being used as a read-only reference to important constants or definitions.

In developer interviews we also revealed an interest in visual clues about how important *AutumnLeaves* considers a window. This not only supports developers in estimating when a particular window will be removed, but is also helpful in locating windows still being actively used. *AutumnLeaves* currently visualizes the internally maintained weighting of windows by showing in the window title bar different colors. For windows considered to be active (their weight is above the threshold), the title bar is colored in a heat gradient from red to blue, while red means

very important, blue less important, as suggested by other researchers [EICK 92, KERS 05]. Windows identified for closing are grayed out in a gradient from light gray to black, where black indicates a weight far below the threshold. Such visual clues also serve as a navigation aid to developers, as they often find the window of interest by looking at the title bar colors. In most cases an interesting window has a non-gray color and often even a red color.

As future work we leave to empirically determine the impact of different weights (as shown in 5.2 and Table 5.3) in practice, the gain on productivity of *AutumnLeaves* or the correlation between the window importance computed by *AutumnLeaves* and what developers themselves consider to be important windows.

5.4.3 Differences between IDEs

As the two IDEs, Eclipse and Smalltalk, have fundamental differences in their window management (as mentioned in Section 5.2), we also expect differences regarding *AutumnLeaves*. Even though the Eclipse data sets do not significantly differ from the Squeak data sets, the low number of Eclipse data sets (3 for Eclipse compared to 22 for Squeak) does not allow us to draw a statistically relevant conclusion. Generally, the *AutumnLeaves* algorithms are less complex in Eclipse as for instance visibility is just a boolean variable — either a tab is visible in the tab bar or it appears in the tab list (making it essentially invisible). Moving and resizing of windows is not relevant in Eclipse.

Usage data shows, however, that in Eclipse more windows are open on average (see Section 5.2), probably due to the fact that Eclipse only supports limited navigation in windows (for instance, we cannot open a new class in an existing window). So far we have not analyzed enough data sets from Eclipse to judge whether we have to adapt considerably the *AutumnLeaves* algorithms or the weighting mechanism to adapt to the navigation differences in Eclipse compared to Squeak. The data we analyzed gives us the impression that *AutumnLeaves* is robust enough to also properly handle Eclipse window management. Further work aims at gathering and analyzing more Eclipse development data. At the moment we are optimistic that *AutumnLeaves* requires only fine-tuning of, for instance, the weighting procedure to work equally well in Eclipse as in Squeak.

5.5 Summary of the Chapter

In this chapter we studied the window plague which overloads the workspace and hampers the overview of the maintained software system in most modern IDEs such as Eclipse or Squeak Smalltalk. We analyzed several development sessions of various developers to reveal the extent and graveness of workspaces crowded with many windows. Developers remarked that an automatic means to close windows is beneficial for them and thus we implemented *AutumnLeaves*, a mechanism that observes all open windows and how they are related to each other by associating a weight to each window. This weight reflects the current importance of a window and its content (classes or methods) and thus *AutumnLeaves* can identify obsolete windows that are most likely not useful anymore in the current development session. *AutumnLeaves* automatically closes such windows, if developers do not decline this. We evaluated *AutumnLeaves* with a benchmark validation analyzing 25 recorded development sessions to determine the correctness of *AutumnLeaves*' algorithms. The correctness results reveal that *AutumnLeaves* is usually able to pinpoint the windows that are appropriate candidates for closing. We further reported on the practicability of our approach and critically discussed it.

With *AutumnLeaves*, we contribute an approach to reduce the amount of information with which developers have to deal in IDEs by automatically closing windows. *SmartGroups* and *HeatMaps* pursue a similar goal by providing a working context in which developers just need to consider a reduced set of entities or by emphasizing the entities of interest in the entire source space. In the next section, we present some other approaches giving a better overview of a system to developers working in the IDE and aiding them to easier locate important entities in a large software space. The following section also studies the achievements of the entire first part of this dissertation and identifies IDE issues not yet addressed.

Chapter 6

Discussion

We first briefly look at some other techniques we implemented to alleviate the information overload IDEs before we conclude the first part of this dissertation by critically discussing all so far presented proposals and by identifying IDEs problems still not properly addressed.

6.1 Other IDE Enhancements Tackling Information Overload

The previously discussed approaches do not completely solve the information overload problem in IDEs. In particular, gaining a quick, higher-level overview of the system is still not easily possible. To tackle the problem of missing higher-level overview, we integrated several visualizations in the IDE such as a system complexity view, class blueprints, and UML diagrams.

Another IDE enhancement we contribute places icons next to source artifacts to show information not directly visible in the static software structure, such as whether a method is overridden in subclasses or whether a message has any senders.

These two enhancements further mitigate the problem of being overloaded with too much information in IDEs by supporting developers in finding their way in a large software space. Both visualizations and iconic information guide developers by highlighting important artifacts that need further attention and thus help developers to stay oriented even in an overloaded workspace. As these two enhancements are not

novel scientific contributions but rather the integration of existing works and ideas into development environments, we placed the discussion of this work in the appendix (Chapter A) where we elaborate on how we concretely integrated these two techniques in the IDE.

6.2 Conclusions

We summarize our contributions and evaluate how they address the information overload problem. Afterwards, we identify the shortcomings of IDEs that are not or only partially addressed by the aforementioned three proposals (*HeatMaps*, *SmartGroups*, and *AutumnLeaves*). These unsolved issues of IDEs motivate us to propose the approaches presented in the second part of this dissertation.

6.2.1 Problems Addressed

Figure 6.1 summarizes all problems addressed in the first part of the dissertation and reports on the development activities that are now better supported.

Problem \ Activity		Feature investigation	Feature implement-ation	Artifact investigation	Dependency investigation	Runtime interaction investigation	Artifact usage investigation	Execution pattern investigation	Quality assessment	Domain concept analysis
Overloaded views	Information overload	HM SG AutumnLeaves	HM SG AutumnLeaves	HM SG AutumnLeaves		HM SG AutumnLeaves				HM SG AutumnLeaves
	No overview	HM AutumnLeaves	HM AutumnLeaves		HM AutumnLeaves				HM AutumnLeaves	
	No context, task support		HM SG							
Narrow focus on static source perspectives and views	Distributed artifacts	HM SG		HM SG	HM SG	HM SG		HM SG	HM SG	HM SG
	Collaboration hidden	HM SG		HM SG	HM SG	HM SG		HM SG	HM SG	HM SG
	Execution paths hidden			HM	HM	HM	HM	HM		

Figure 6.1: The various IDE shortcomings addressed by the proposals presented in the first part of the dissertation (*HeatMaps*, *SmartGroups*, and *AutumnLeaves*) and the development activities to which these proposals contribute (HM = *HeatMaps*, SG = *SmartGroups*).

Information overload, missing overview. All three approaches address the problem of being overloaded with too much information and lack of overview in the IDE: *HeatMaps* highlight in the entire software space those entities with a high probability of being important for the current task to better gain overview and focus in a large source space. *SmartGroups* allow

for automatic or manual categorization of interesting source artifacts and thus help developers to focus on a subset of the entire, overloaded source space. Finally, *AutumnLeaves* reduces the number of open windows to better maintain overview and focus in the surfeit of windows developers usually have to open even during relatively short development cycles.

Missing context and task support. In particular *SmartGroups* contribute to the problem of missing representation of context in the IDE as they enable developers to manually build such a context by categorizing source entities or by automatically constructing a task context based on how the system has been previously modified, navigated, or updated. To a limited degree, *HeatMaps* represent context by applying a heat color scheme to all source entities. All artifacts colored in a hot color can be considered as being part of a particular context, such as being recently modified for the correction of a bug.

Hidden collaboration between distributed artifacts. Eventually, both *SmartGroups* and *HeatMaps* aid developers in identifying communication and collaboration patterns between distant and distributed artifacts, such as classes dependent on each other but located in different packages of a software system. *SmartGroups* categorize such distant artifacts in the same smart group, for instance when the artifacts are changed or navigated in tandem. *HeatMaps* color them in the same heat color, thus both techniques make obvious that these distant artifacts are conceptually related, hence they might collaborate with each other at runtime. However, neither *SmartGroups* or *HeatMaps* make such collaboration directly visible, the explicit representation of hidden collaboration hence remains unachieved.

Execution paths hidden. As *HeatMaps* are also able to use runtime information for the heat coloring of source artifacts, a map could for instance color all entities used in a particular execution scenario. The more often a method is used, the more red it is colored by this map in the IDE. Such an usage of *HeatMaps* highlights to a limited degree the execution paths occurring in a system, namely the actually executed entities. However, *HeatMaps* are not able to reveal relationships between the executed artifacts, for example the execution order. We thus present in the second part of this dissertation more elaborated means to visualize execution paths, in particular also on a source code level.

6.2.2 Remaining Problems

Several problems of IDEs as identified in Section 1.1 cannot be addressed by *SmartGroups*, *HeatMaps*, or *AutumnLeaves*. Some of these unsolved problems are best tackled by exploiting dynamic information in the IDE. In the following, we briefly study which problems remain unsolved by the three techniques presented in this first part of the dissertation.

Quality assessment support poor. The three techniques *SmartGroups*, *HeatMaps* and *AutumnLeaves* do not aid developers in assessing the quality of the software system they are developing. In Chapter 8, we present *Senseo* which offers limited support for quality assessment by emphasizing in the IDE source artifacts that consume much memory or create many objects. As stated in Section 1.1.2 this work does not aim at thoroughly supporting software quality assessment from within the IDE.

Hidden collaboration between distributed artifacts. While *SmartGroups* and *HeatMaps* offer some limited support to identify conceptually related but distributed artifacts, these two techniques are not able to make explicit the dynamic collaboration patterns between such distant artifacts. In the case of an unfamiliar system with no recorded history of previous navigation or modification occurring in the IDE, *SmartGroups* and *HeatMaps* are usually not able to detect distant artifacts. We contribute several techniques to better determine collaboration between distributed artifacts and to better embed identified collaboration patterns in the IDE. *Hermion* (Chapter 7), *Senseo* (Chapter 8), *CollView* (Chapter 9), and *FeatureEnv* (Chapter 10) all contribute with different means to the problem of distributed, distant artifacts whose collaboration is not obviously visible in the static perspectives of traditional IDEs.

Execution paths hidden. With *Hermion* (Chapter 7), *CollView* (Chapter 9) and partially also with *Senseo* (Chapter 8), we contribute techniques that allow developers to study execution paths in a software system on a high package level and on a low method or even source code level, for instance which method invokes how often and with which arguments which other methods, or which branch of an if statement is executed how often.

Imprecise static source code. *SmartGroups*, *HeatMaps*, and *AutumnLeaves* do not address the problem of not having precise information in the static source code about how it behaves at runtime, such as runtime type information. Hence, we propose with *Hermion* an approach tackling this problem. We discuss *Hermion* in detail in Chapter 7.

Features hidden in code. With the three techniques from this first part of the dissertation, it is not possible to directly reason about software features as first class entities. Manually created smart groups could theoretically contain all source artifacts implementing a specific feature, but *SmartGroups* do not support the automatic creation of such feature groups. This issue is addressed with *FeatureEnv* which enriches the IDE with an explicit representation of features by visualizing them in interactive views. Chapter 10 discusses *FeatureEnv* in detail.

In the following second part of the dissertation, we study in detail the four techniques addressing the aforementioned problems. These four techniques, *Hermion*, *Senseo*, *CollView*, and *FeatureEnv* have in common that they gather, analyze, exploit, and integrate by various means dynamic information about the system under study in the IDE.

Part II

Exploiting Dynamic Information in IDEs

The second part of this dissertation aims at tackling the narrow focus of IDEs on static software structure by integrating dynamic information into the static source perspectives of IDEs in order to improve system navigation and understanding, in particular of dynamic collaboration between scattered and distributed code.

We present four distinct approaches to embed various kinds of dynamic information in the familiar IDE views and tools.

- *Hermion* (Chapter 7) augments the understanding of static source code by enhancing an IDE's source code views with dynamic information such as runtime types of variables or receiver types of message sends.
- *Senseo* (Chapter 8) integrates into the static source perspectives information about dynamic collaboration between methods, classes, or packages and thus improves system overview, navigation, and understanding.
- *CollView* (Chapter 9) embeds in the IDE interactive, navigable visualizations of dynamic collaboration patterns between methods, classes, and packages, in particular to enhance the overview of the runtime behavior of a system.
- *FeatureEnv* (Chapter 10) explicitly represents software features in the IDE by providing visualizations showing all exercised source artifacts during a feature's execution to improve feature understanding and navigation.

We conclude this second part of the dissertation by discussing these four approaches with respect to the tackled problems regarding IDEs narrow focus on static software structure. We also look back at the first part to summarize which IDE problems and shortcomings we successfully tackled in the entire dissertation.

Chapter 7

Hermion – Extending Source Code Perspectives with Dynamic Information

7.1 Introduction

7.1.1 Positioning *Hermion*

In this chapter we present *Hermion*, a technique integrating dynamic information into the source code perspectives of the Squeak and Pharo Smalltalk IDE. *Hermion* particularly aims at improving the understanding of unclear and imprecise static source code, thus this approach enriches the source code views with runtime types of variables or receiver and argument types of message sends. Additionally, *Hermion* presents all the types dynamically referenced by a particular source artifact during the recorded execution, for instances all classes that have been used in a method.

Figure 7.1 summarizes all of the IDE problems introduced in Chapter 1 that *Hermion* tackles. Most notably, *Hermion* alleviates the problem of unclear static source code presented in an IDE's source code views by enhancing them with dynamic information. The availability of such behavioral information in the source code views also helps developers to reconstruct the intra-procedural execution flow in methods, particularly as *Hermion* highlights all executed statements. Moreover, *Hermion* also

	Problem \ Activity	Feature investigation	Feature implement-ation	Artifact investigation	Dependency investigation	Runtime interaction investigation	Artifact usage investigation	Execution pattern investigation	Quality assessment	Domain concept analysis
Static views	Distributed artifacts	✔		✔	✔	✔		✔	✔	✔
	Collaboration hidden	✔		✔	✔	✔		✔	✔	✔
	Execution paths hidden			✔	✔	✔	✔	✔		
	Imprecise static source code			✔		✔		✔		

Figure 7.1: *Hermion* primarily addresses the problem of imprecise static source code and also of unclear execution flow in methods. Additionally, hidden collaboration between distributed artifacts is made explicit on a method and class level.

contributes to a better understanding of hidden collaboration between distributed artifacts, mainly because it presents all types referenced in an artifact. Thus, *Hermion* basically contributes to all development activities apart from feature implementation.

In the remainder of the chapter, we describe our *Hermion* proposal. We introduce the research questions to be tackled by *Hermion*, motivate the need for dynamic information embedded in source code perspectives, describe in detail how we integrated runtime information in the Smalltalk and how we gathered it, and finally validate the approach by means of an efficiency benchmark and an empirical evaluation with developers.

7.1.2 Introduction to *Hermion*

Gaining an understanding of large object-oriented systems by navigating the source code in a development environment (IDE) is an inherently difficult and time-consuming task. Object-oriented language characteristics such as inheritance and polymorphism make it difficult to understand how an application is implemented purely by navigating and browsing source code [DEME 03, DUNS 00, WILD 92]. Often conceptually related code is scattered over many different source artifacts, for example classes and methods. The task of program understanding is more acute with dynamically-typed languages such as Smalltalk or Ruby, as developers typically require access to runtime type information to gain a complete understanding.

Program comprehension is a prerequisite when faced with the task of extending and maintaining a system. Exploration of a system is constrained by the mechanisms provided by the IDE to browse and navigate a large software space. However, as discussed in Section 1.1.2, today's popular IDEs base browsing and navigation mechanisms only on a sys-

tem's static source code. They do not provide an integrated view of the dynamic and static structures of the system, but narrowly focus on the static view only. They offer little to understand the runtime behavior of a system. Researchers in program comprehension have recognized the value of combined static and dynamic views for program comprehension [DEME 00, DUNS 00, LÖWE 01, SYST 99, WILD 92]. But nonetheless IDEs do not usually present any dynamic information in their source perspectives.

In this chapter we identify different kinds of dynamic information and illustrate how each contributes to a developer's system understanding. We claim that direct access within an IDE to runtime information such as message sends, class references and runtime types of variables enhances program comprehension through informative and efficient browsing and navigation. It is crucial that dynamic information is embedded in the IDE without further overloading the already busy interfaces and without forcing developers to learn new and complex means to access this dynamic information. The key research questions we address in this chapter are:

- *How do we integrate dynamic information into an IDE's browsing and navigation mechanisms without overloading the existing source views even more?*
- *How can we efficiently collect dynamic data in a running IDE?*

We address these questions and present our working prototype, an IDE called *Hermion* with the capability to capture runtime information from an application under development and to exploit this information by enhancing navigation and browsing of the source code. We validate the usefulness of *Hermion* by using it to understand two medium-sized applications. As validation of our work we perform a preliminary experiment where we ask five developers, unfamiliar with the applications, to report on how *Hermion* supports their understanding of the systems.

We cover and address the typical dynamic analysis issues such as efficiency, coverage and completeness and outline future improvements.

The key contributions of this chapter are: (i) we identify which kinds of dynamic information are useful for enhancing the navigation and understanding of systems in the IDE, (ii) we describe our dynamic information enhancements to the IDE, and (iii) we apply a partial behavioral reflection technique to selectively gather runtime information within the IDE. The goal of this chapter is to report on a means to improve the understanding of imprecise static source code, of execution flow in such code, and of hidden references (that is, classes being dynamically referenced in a method or another class).

Outline. In the next section we identify shortcomings of purely static IDEs and elaborate with example scenarios on what kinds of dynamic information can improve and optimize the navigation of a source space. Section 7.3 presents our technique to dynamically collect runtime information. We present the validation of our work in Section 7.4 with two medium-sized object-oriented systems and a preliminary empirical evaluation. In Section 7.5 we discuss our work by highlighting efficiency, coverage and completeness issues. Section 7.6 presents related work, while in Section 7.7 we draw our conclusions and outline future work.

7.2 Dynamic Information in the IDE

This section answers the research question: *How do we integrate dynamic information into an IDE's browsing and navigation mechanisms without overloading the existing source views even more?*

We motivate our work by highlighting the restrictions a developer faces in IDEs when trying to understand systems implemented in object-oriented and dynamically-typed languages. The general problem is that the IDE's view focuses purely on static source code. It provides little support for understanding the dynamic behavior of a software system, in particular of systems written in dynamic object-oriented languages that make widespread use of inheritance and polymorphism. This makes it difficult for a developer to determine how classes interact at runtime, for instance to which receiver a message is sent at runtime, or what kind of objects are stored in variables. In dynamically-typed languages such as Smalltalk or Ruby, but also to a lesser degree in statically-typed languages such as Java, the IDE generally provides no means to support the developer browsing the source code to understand the runtime behavior of the system under study. For instance, the Eclipse IDE [ECLI 03] does not provide an integrated view showing the precise types that are dynamically assigned to a variable or to seamlessly navigate directly in the source code to the message sends actually occurring at runtime.

7.2.1 Scenario: Understanding a Complex System

In the following we illustrate with a scenario some concrete problems a developer faces when trying to understand a complex and generic application written in the dynamic language Smalltalk. Subsequently, we present our experimental "dynamic IDE" called *Hermion* which encompasses solutions to the problems we are going to identify. We emphasize

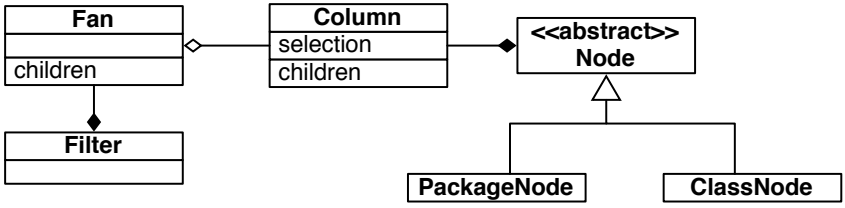

Figure 7.2: UML Class Diagram of the OmniBrowser kernel classes.

that the issues we address in this discussion are generally applicable in the context of IDEs and object-oriented and dynamically-typed languages.

We center our discussion around an example software, the OmniBrowser framework. OmniBrowser is a highly customizable, generically implemented framework, rendering it well-suited for the implementation of a range of source code browsers (*e.g.* package browsers or class browsers). It makes extensive use of method dispatching and polymorphic message sends which may make it difficult to understand its source code purely by static browsing.

We briefly introduce the OmniBrowser framework, providing a structural overview with a UML class diagram in Figure 7.2. It provides a class Column to represent a vertical list of selectable elements, for instance a list of packages. Every element in the column is an instance of class Node. Concrete types of nodes are represented by subclasses of Node, for example ClassNode or PackageNode. The elements of a column may be filtered. This feature is realized using a dedicated Filter class. The Fan class manages the different filters dynamically applicable to a specific column.

We describe a realistic scenario of how a developer, new to the OmniBrowser framework, might gain an understanding of the code. Initially, the developer wants to understand how elements are loaded into a column for display. Subsequently, she tries to discover how a selected element is represented in a column. Finally, she wonders how different columns are embedded in a browser.

Message Send Navigation. To find out how OmniBrowser loads elements in a column for display, the developer may start by looking at the class Column. This class defines a method children which reads as follows:

```
children
    ^fan children
```

Browsing this method reveals that Column delegates the message send children to an object called fan which then apparently answers all elements of a column. Static browsing does not reveal where the elements actually come from. By just looking at the source code of Column»children it is not obvious which children method will eventually return the elements. Furthermore, the class of the variable fan cannot be determined statically. Due to polymorphism, different children methods may be invoked at runtime, depending on what kind of object is stored in fan. Often modern IDEs, such as the Squeak IDE, provide an *implementors view* which displays a list of all methods named children that exist in the current application. However, this mechanism populates the list of methods by performing a static search of the entire application for children methods. This results in a long list of possible methods (see Figure 7.3). Exploring this list is time-comsuming and inefficient.

When we analyze the runtime behavior of an OmniBrowser application, we discover that only one single children method, the Fan»children method, is invoked from the children method of the Column class. This implies that if the IDE provided a mechanism to consult the runtime information, the search space for the developer could be greatly reduced, thus resulting in a more precise and efficient navigation of the source code. Figure 7.3 compares the different lists of implementors, on the left is the list generated using the existing *implementors view* mechanism (1) and on the right is a list that was generated by *Hermion* which takes dynamic behavior (that is, message sending) of the OmniBrowser application into account (2).

The IDE could even include the navigation of message sends directly in the source code view by enriching this static view with dynamic information gathered while a method is being executed. In the Column»children method for instance we can annotate the message send #children to the fan object with an icon. Clicking on the icon while reading the method's source code navigates the developer directly to the implementation of the children method in the class Fan, which is the only method invoked at runtime at this location of the code. In the case of a polymorphic message send which has different receivers, there are often several possible methods that may be executed. In such a case, clicking on the icon near the written message send results in the display of a list of all methods that have been invoked at runtime. This list also contains information about the receiver type of the message send and which and how often a method was invoked. In Figure 7.4. we present an example of this method list

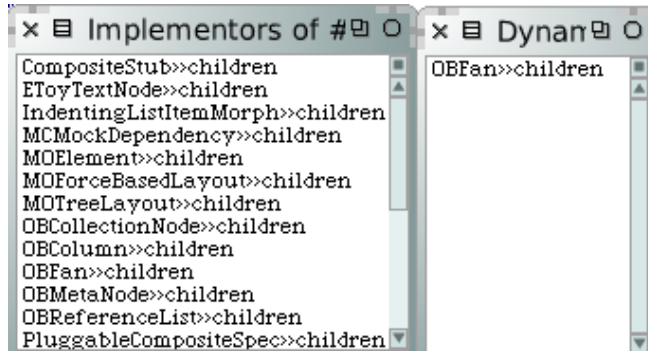

Figure 7.3: Static search (1) vs. precise dynamic search (2) for implementors of *children* in *Hermion*.

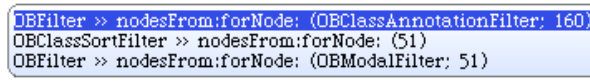

Figure 7.4: List of methods invoked for message send *nodesFrom:forNode:* in *Hermion*.

in *Hermion* for the polymorphic message send #nodesFrom:forNode: which has two different receiver types.

Type Information. The developer wants to know how a column stores the selection of a specific element. As the OmniBrowser framework is implemented in Smalltalk, type information is not available in the source code. Generically implemented frameworks usually suffer from the same problem even in statically typed languages as they refer to abstract classes in the source code, which makes the source code hard to understand as at runtime concrete classes are used. Thus browsing the source of method selection does not reveal the type of information the selection instance variable contains at runtime (see Figure 7.5). It may contain the selected element itself (that is, a node) or the index in the list of elements (that is, an integer). However, such type information is essential to understand how the application generates the list of new elements for the subsequent column.

By analyzing the selection method of Column dynamically, an IDE can provide the developer with precise information about how the current selection of the column is stored, information that is otherwise hard to determine, when variables can virtually store any kind of object.

Figure 7.5: List of types of instance variable *selection* extracted from dynamic information in *Hermion*.

Similar to the message send icon previously described, *Hermion* also enhances the static view of variable accesses in the source code with an icon. Clicking on this icon reveals a list of object types that have been stored in this variable at runtime at this position in source code, along with quantitative information, for example the number of times this variable has assumed a specific type. We refer to this mechanism of revealing the type of variable as *type view*. To promote deeper understanding of the code under investigation, a developer may wish to use the type information of variables as starting points for further navigation to the corresponding classes behind the types.

Figure 7.5 depicts the *type view* for the selection instance variables in the method selection. It reveals that the selection variable always stores an integer, which in turn reveals the intent of the Column class to store the position of the selected element in the list, not the element itself.

Reference Information. If a developer wants to add a new column to a browser based on the OmniBrowser framework, but is unsure how to do this, static browsing of the code of class Column does not quickly reveal which class implements the container component for the columns, as this communication is very well hidden in a few methods of Column.

Using runtime information, we can identify all classes with which Column communicates. We refer to those classes as dynamic references of Column. Many current IDEs such as Eclipse are only capable of locating those references that are explicitly written in source code. As our example shows, the dynamic references are more interesting and useful for understanding. By integrating type inference mechanisms, the IDE could provide a mechanism to generate a more comprehensive list of referenced classes. For dynamic languages this list is still often not accurate or even correct, because type inference in dynamic languages is not able to infer the types correctly in all situations [RAPI 98].

If we analyze a software system dynamically we can provide a precise and correct list of references in a class. We propose a *reference view* that presents a list of referenced classes based on dynamic information. This view reveals to the developer which classes are referenced by the class under investigation, for example in which methods and in which vari-

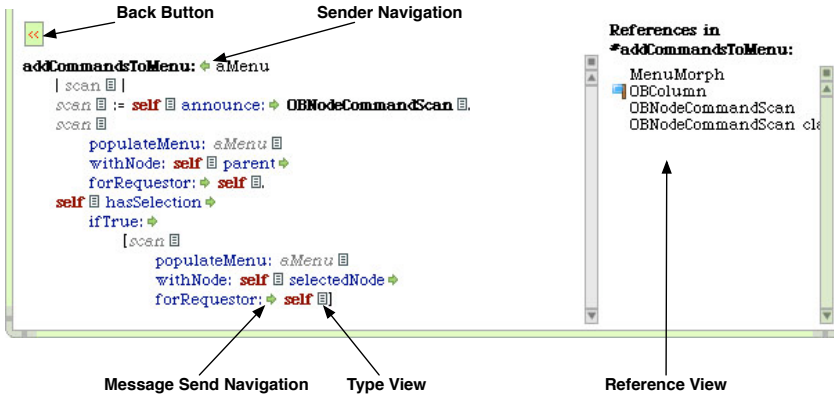

Figure 7.6: Enriched method source code view including a reference view in *Hermion*.

ables. Similar to our proposed *type view*, the reference view is enriched with quantitative information extracted from the dynamic information, for instance how often classes are referenced. When the developer selects a class or a specific method of this class, she is immediately provided with this reference view (see Figure 7.6 on the right) to learn to which other class the selected class or method communicates. In this reference view the developer also finds a class called ColumnPanel whose intention revealing name indicates that it contains the different columns the OmniBrowser holds. By looking at the places in Column where this class is referenced dynamically, we indeed learn that columns are added to an instance of ColumnPanel when the browser is built up.

7.2.2 Hermion Overview

As a proof-of-concept for our claims, we implemented *Hermion* as an enhancement to the Squeak Smalltalk IDE [SQUE 10] to integrate the collection and presentation of dynamic information. Our solution is not restricted to Smalltalk or Squeak. The mechanisms described in this chapter and the problem of gathering dynamic data in the IDE are applicable to any of the widely used IDEs such as Eclipse [ECLI 03] .

Our approach to seamless integration of dynamic information in IDEs is based on enriching the mechanisms and tools for navigating, browsing, viewing and editing source code currently existing in the IDE. The developer does not need to change her perspective or learn a new tool. She is just provided with new features in her familiar environment. In the case

of the source code view where the developer browses, writes and edits source code, we enriched this view with additional visual information (based on the use of color-coding) that takes the runtime behavior of the application into account. This visual information is presented in a lightweight approach by using small icons that are supposed to avoid overloading the source views even more. In Squeak Smalltalk the developer browses source code in a class browser which currently displays a single method at a time. Figure 7.6 shows an example of a enriched method view for method addCommandsToMenu: of class Column. Variable accesses are shown in gray, message sends in blue. Each statement is also enriched with a clickable icon to access the gathered dynamic information.

In this enriched source code view we integrated the solutions to two of the mentioned problems in Section 7.2.1: message send (as well as sender) navigation and type view are integrated in the source code view while the reference view is placed to the right of the source code pane.

Such an enhanced IDE is not only useful for maintaining existing software systems, but shows its advantages also during development of new software. The developer can for instance continuously dynamically analyze her application under development and see in the IDE if a method invokes the methods it is indented to invoke, or if classes communicate correctly to other classes at runtime. This gives the developer viable feedback about the dynamics of her program even during development.

7.3 Dynamic Information Gathering

In this section we elaborate on the question: *How can we efficiently collect dynamic data in a running IDE?*

As already discussed in Chapter 2, traditional approaches to collecting dynamic data such as tracing are generally not efficient enough to provide dynamic data to be displayed at runtime in the IDE. Another drawback of these approaches is that they typically yield large amounts of data, making it difficult to mine useful information [CORN 07a]. Thus it is not really feasible to adopt these approaches as a basis for providing direct access to dynamic information in an IDE.

With *partial behavioral reflection* [TANT 03] we found a means to efficiently gather dynamic data as this approach enables selective collection of a system's runtime data. Dynamically analyzing the entire system is usually not necessary, because during the maintenance phase a developer is mainly interested in specific parts of the system, for instance in the classes implementing a specific feature. Furthermore, our approach does not generate a huge traceing file intended for offline postmortem analysis,

but directly gathers and stores the data in the format in which it can be used without any offline analysis. This format is optimized for fast lookup, that is, it contains precisely the information the IDE wants to present in the form of objects.

7.3.1 Partial Behavioral Reflection

After having briefly introduced *Partial Behavioral Reflection* in Section 2.2.2, we analyze this approach based on a concrete implementation called Reflex [TANT 03] in the following in more detail. Reflex offers mechanisms to precisely select entities (*e.g.* classes, methods, variables) and operations (for example, message sending, variable access). In particular, the approach of Reflex allows us to collect data at different levels of granularity, also at a sub-method level [DENK 07]. For instance, we can select what kind of operation occurrence in a method we want to reflect on (for instance, only accesses to a specific variable) and can also precisely determine what information has to be reified about this operation (for example, only the value and name of a variable). Hence the Reflex model is well suited to efficiently gather runtime information about the program currently being developed [DENK 06]. For our work we use a variation of Reflex in Smalltalk, called Geppetto [DENK 07, RÖTH 07d].

The crucial entity in the Reflex model is the *link*. A link *causally connects* the base level of a system with its metalevel. Links are introduced in the executable of an application and upon occurrences of base-level operations they invoke methods on the corresponding metaobjects. During the installation of reflective behavior, the developer precisely selects the operations where links are to be installed and specifies the metaobject that the link will trigger. In our scenario, this metaobject implements a tracing mechanism which stores dynamic data captured at runtime in a format that is accessible to the IDE. The concepts of base level, link and tracing metaobject are illustrated in Figure 7.7. The "x"s in the application code are messages sends to be traced at runtime. For every message send we install a link that is triggered at runtime. This link reifies the runtime information about the occurring message send and passes this information to the tracing metaobject which can then reason about this information, for instance store it in a database. For a more comprehensive treatment of partial behavioral reflection we refer the reader to the work of Tanter *et al.* [TANT 03].

To gather dynamic data required to realize the features presented in Section 7.2, we install three links for every method to be analyzed: The first link reifies receiver and arguments of every message send, the second link intercepts variable accesses by reifying name and value of

a variable while the third link does the same, but for assignments to variables where also the new value of the variable is reified. All these links pass their information to a meta-object where it is processed and then stored in a database, and optimized for efficient retrieval so that the dynamic information can be readily embedded in the IDE at various places. We eliminate the need to gather large traces, thus minimizing the amount of dynamic data necessary for runtime analysis of the system.

We sum up by emphasizing the advantages of using partial behavioral reflection over trace-based approaches:

- *Precise specification of the kind and amount of dynamic information.* The developer controls when and where dynamic data is gathered, even while the application under study is running. The selection is driven by a developer's current need. If a specific software feature contains a defect, the developer selects the feature's source elements to be analyzed dynamically. The developer's selections directly influence the performance and amount of collected data, thus the developer can control how efficiently the data gathering will run.
- *Instantaneous access to accurate runtime information within the IDE.* Runtime information is processed while it is being gathered (that is, while the application is running). The selected source entities are instrumented to introduce the reflective behavior necessary for gathering dynamic information. Instrumenting a medium-sized class typically takes less than a few milliseconds. On subsequent execution of these entities, dynamic information is collected and displayed in the IDE. For instantaneous integration of runtime information in the IDE it is crucial to apply a fast data collecting technique such as partial behavioral reflection so that dynamic information is readily available in the IDE.
- *Mapping of dynamic data to static source elements.* Partial behavioral reflection supports dynamic data gathering about sub-method elements such as message sends or variables accesses. This gathered data can easily be mapped back to the specific declarations of message sends or variables accesses in source code to make sure that the developer sees the dynamic behavior which occurred at a specific location in code.

7.4 Validation

To validate that our approach is usable and scalable in real-world scenarios, we applied it to two case studies. As a first case study we chose Pier,

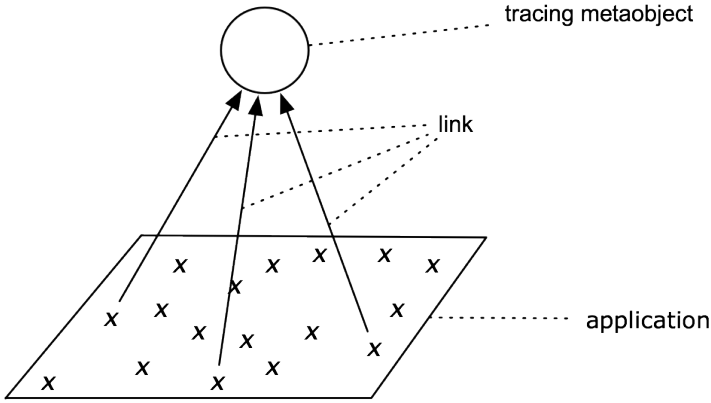

Figure 7.7: The link invokes the metaobject upon occurrence of selected base-level operations.

a web-based content management system written in Smalltalk [RENG 07]. As a second case study, we selected the OmniBrowser framework introduced in Section 7.2, which forms part of the traditional IDE of Squeak Smalltalk 3.9 [SQUE 10]. We performed benchmarks using these two applications to report on efficiency issues, that is, the factor by which the applications slow down the IDE when gathering dynamic data and when presenting dynamic information. Subsequently, we present results of a preliminary empirical evaluation where we asked developers unfamiliar with Pier and OmniBrowser to test our environment while working with these systems.

7.4.1 Case Studies: Pier and OmniBrowser

Pier [RENG 07] consists of 104 classes, most of which are implemented generically. This makes it quite hard for a developer to understand for example how these generic classes interact, that is, which methods invoke which other methods of other classes at runtime. For that reason, we consider Pier to be an appropriate case study to evaluate the usefulness of our enriched IDE. We asked developers unfamiliar with Pier if our enriched IDE supported them in gaining an understanding of the application and if it eases navigation of the application classes. Before we present the results of this empirical evaluation in Section 7.4.3, we first verified that is is technically possible to instrument Pier and to integrate the resulting dynamic data in our IDE.

	Not instrumented (ms)	Message sends only (ms)	Message sends and variables (ms)
Pier, core classes	42	141 (+236%)	297 (+607%)
OmniBrowser, core classes	625	2343 (+274%)	4146 (+564%)

Figure 7.8: Comparison of execution times for different levels of instrumentation for OmniBrowser and Pier.

In our initial experiment, we only instrumented the seven core classes of the Pier model package. While the dynamic data gathering was active in these core classes, we subjectively noticed just a negligible slowdown of the running Pier application.

In a second step, we instrumented the entire Pier application. This instrumentation only takes a few seconds to perform. Even though the application slowed down while dynamic data collection was active, we were instantaneously able to make use of the collected information in the IDE. We consider this as a good indication that it is possible to access and use dynamic information in an IDE while a system under study is running.

OmniBrowser [BERG 07b] is a framework to create various kinds of browsers in Smalltalk. It currently consists of approximately 170 classes. We also used this framework to implement *Hermion*. With this experiment, we analyze the OmniBrowser framework dynamically within itself. By displaying the results of the dynamic analysis, we trigger the generation of new dynamic information which is instantly available in the IDE.

We instrumented all eleven classes in the model package of OmniBrowser. These classes form the core of OmniBrowser and are heavily used. We were able to successfully instrument all these classes to gather information about message sending and variable accesses occurring at runtime. Despite partial behavioral reflection instrumentation, the active browser in the IDE was still usable even during the collection of the dynamic information. We were able to immediately look at the dynamic information collected, for instance we could navigate the message sends or view the types of variables as explained in Section 7.2.2. We verified that our technique did not introduce flaws in the functionality of OmniBrowser, as we could still successfully run the comprehensive test suite of OmniBrowser. We were able to accurately collect all the required dynamic information even for methods heavily used at runtime.

7.4.2 Efficiency

As a performance test we compared the execution speed of the two applications, Pier and OmniBrowser, with and without dynamic data collection. Furthermore, we varied the extent of data collection: In a first scenario, we reified information about message sends, while in the second scenario we also reified variable accessing. As a reference, we measured the execution time for the same usage scenario while no data collection is active.

In Pier, we measured the time to display the standard start page of the web application. In the case of OmniBrowser, the usage scenario of the application was opening a new system browser which included our integration of dynamic information. Figure 7.8 contains the results of this performance test. The slowdown introduced due to the collection of dynamic information is perceivable, in case of the Pier framework we observe a slowdown of factor 7. We noticed that it makes a difference of more than factor 2 if only message sending or also variable accessing is reified. This represents clear evidence that the amount of information reified and collected at runtime has a high impact on performance. Limiting the number of reifications, for instance only one kind of operation, is a good strategy for improving performance. The same is true for the coverage of static source artifacts: It is much more expensive to cover all classes, that is, to gather dynamic information about the usage of all classes in a package, than to select precisely the classes for which dynamic information is actually of interest.

In Section 7.5 we discuss the impact of these performance results on the usefulness of *Hermion* in a practical situation where it is used to comprehend or maintain a software system.

7.4.3 Preliminary Empirical Evaluation

We conducted an empirical evaluation with five Smalltalk developers to learn how useful and practical they consider *Hermion*. The subjects used *Hermion* to learn how the Pier application is structured. Concretely, we gave them the task to understand how a page is stored and displayed in Pier. None of the subjects had ever worked with Pier before, but were familiar with the Squeak IDE. We present qualitative feedback the developers gave us as well as our observations made while watching the subjects working with *Hermion*.

We presented the subjects with a list of concrete questions about how certain aspects of Pier (for example displaying, adding a new or moving an existing page) are implemented or how certain classes interact with each other. Some of the questions they had to answer using dynamic

information integrated in *Hermion*, the other questions they had to solve in the original Squeak IDE. As a start, we advised the subjects to look at and to dynamically analyze three core classes of Pier. From there on, they could start to understand how specific features of Pier are implemented and how those classes interact with the rest of the system.

Our own observations reveal that the answers of the subjects were significantly more precise when they were working with the dynamic information in the IDE. Concerning efficiency, our measures report that subjects could answer all of the questions in approximately 25 percent less time.

Developer Feedback. We gave the subjects a questionnaire covering various aspects of *Hermion*. In this questionnaire subjects could rate the contribution of *Hermion* on execution overview, type information, navigational aid and program comprehension. We used a five-level Likert scale [LIKE 32] for the ratings ('1' means strongly disagree, '5' strongly agree that *Hermion* has an effect). Furthermore, we allowed the subjects to also write free text on the questionnaire to give us qualitative feedback that cannot be expressed with a Likert scale. In the following we present the results from the questionnaire:

Execution overview. Subjects considered it as very useful (average rating value above 4) to see precisely what parts of the code were executed when Pier is displaying a page. Hence they could focus on the relevant classes and methods necessary to understand how pages are stored and displayed, namely those that are executed at runtime.

Type Information. Subjects stressed that readily available type information makes it much easier to read the source code of a dynamically-typed language (average rating was 3.5).

Navigational aid. Students reported that they could more efficiently locate the methods that were executed by a specific method under investigation when dynamic information is available (average rating was close to 4).

Program comprehension. Consequently, they found that dynamic information accelerated their understanding of the application, as they no longer had to guess what methods would be executed at runtime. In general, all subjects considered the total impact on program comprehension as high. They reported that it took them much more time to comprehend the system when they had to use the original Squeak IDE not providing dynamic information (average rating was above 3).

7.5 Discussion

We discuss some specific restrictions and limitations of our current approach and possible solutions to overcome these limitations. First, we discuss the existing efficiency issues. Subsequently, we elaborate on the coverage and completeness problem of dynamic information in depth.

Efficiency Issues In Section 7.4.2 we spotted certain efficiency issues when analyzing applications dynamically from within the IDE. In the following, we discuss the impact of these results to the applicability of our approach in real-world scenarios and elaborate on optimizations.

During development and maintenance, the execution performance of an application is not crucial. A debugging session in step mode for instance, takes much more time to perform and the developer loses all information visible in the debugging session when she reverts to the source code browsing in the IDE. Moreover, a debugger only shows one distinct run of a software system and neglects information of all other runs. To understand a software system means to effectively navigate the system. For this reason, the dynamic data collection, although not yet fully optimized, still contributes to program comprehension. We consider that the performance of dynamic data gathering compared to the gained benefit does not constitute an obstacle to applying this approach for the task of program comprehension.

Nonetheless, we identified a need to work on these performance issues in the future. Based on our experience with the experiments, a *best practice* for efficiently collecting dynamic information is to only instrument classes which we want to understand to the level of detail required for a given maintenance task. The key advantage of our approach is to provide the developer with the flexibility to choose and define trade-offs of information over performance as she sees necessary.

Coverage and Completeness An open issue is the coverage aspect of dynamic information. The completeness of dynamic information always depends on what is actually executed at runtime. Normally, a full coverage of all behavior of a system is not achievable [BALL 99], instead we only cover parts of the system by, for instance, exercising some specific features. This means that we can only highlight the type information for concrete executions of a system, we cannot show all types that can be theoretically stored in a variable. But this does not need to be viewed as a shortcoming, but rather an advantage. In particular, when maintaining

software, for example correcting defects, the developer is often interested in understanding specific executions of the application, namely those that are broken and have to be revised.

The difference concerning completeness between our approach and analyzing the system in a particular debugging session (that is, using a debugger provided by the IDE) is that in *Hermion* we permanently present all so far gathered dynamic information, this information is the result of various executions of a system and is not volatile, that is, it does not disappear as in the debugger. Furthermore, as mentioned in Section 2.1.4, debuggers included in IDEs such as Eclipse [ECLI 03] or Squeak [SQUE 10] indeed help developers analyzing the runtime of a system, but provide little to support navigation of a source space. In a debugging session, a programmer focuses on a very specific run of the system, observing and analyzing a slice of the program to discover the cause for a specific defect [WEIS 81, ZELL 03]. Our work aims at easing the understanding and navigation of a whole software space in general, the dynamic information thereby presented in the IDE is not restricted to a specific run of a program. Instead we merge dynamic information into the static perspective on source code presented in the IDE.

7.6 Related Work

We relate *Hermion* to other, comparable proposals addressing similar IDE problems as identified in Section 1.1.2. We hereby differentiate between proposals encompassing dynamic information to address the problem of hidden collaboration, unclear static source code and execution flow.

7.6.1 Techniques Encompassing Dynamic Information

To the best of our knowledge, there is not much research in the area of integrating results of dynamic analyses into development environments. However, the work of Reiss [REIS 03] visualizes the dynamics of Java programs in real time such as the number of message sends received by a class. These visualizations are not tightly integrated in an IDE though, but are provided by a separated tool. Therefore, it is not directly possible to use these analyses while working with source code. We consider it to be crucial to incorporate knowledge about the dynamics of programs into the development environment to ease navigating within the source space. Löwe *et al.* [LÖWE 01] followed a similar approach by merging information from static analysis with information from dynamic analysis

to generate visualizations. However, their work is not integrated into the development environment.

Ferret [DE A 08] allows developers to formulate queries to reveal information about how source artifacts are related to each other. These queries also reason about dynamic information, for instance, to determine the exact callers of a method. Thus, Ferret is one of the few proposals to integrate dynamic information in the static source code views of IDEs. As *Hermion*, Ferret is able to identify the methods actually invoked at a method call site. Ferret also addresses the problem of imprecise static source code and unclear execution flow in methods. However, the information about the dynamics in a method are not displayed next to the method source code but separated in a view holding the results of different queries, which makes understanding of source code more cumbersome than in *Hermion* where the information is readily available in the source code view. Additionally, Ferret does not gather and query runtime type information.

Compass [LIEN 09] is a back-in-time debugger for Smalltalk. It is available in the IDE, but shows dynamic information separated from the conventional source code perspectives and browsers. While it also presents information about runtime types of variables or receiver types of message sends, it focuses on a specific system execution while *Hermion* embeds dynamic information aggregated over several executions in the source code views. Furthermore, the dynamic information extracted by *Hermion* is permanently available in the IDE while Compass's information is volatile, only existing during a debugging session.

7.6.2 Techniques Purely Based on Static Analysis

Fluid source code views [DESM 06] augment the source editor of an IDE in similar ways as *Hermion* does. These views also insert clickable icons in the source code editor to show more information. While *Hermion* links source artifacts used at runtime in the source code editor, fluid source code views embed remote method definitions directly in the editor below the declared invocation of these methods. However, fluid source code views do not exploit dynamic information to relate source artifacts, instead they statically link methods to call sites in source code. Thus, for polymorphic call sites fluid source views cannot guarantee to link and embed the correct method definitions. *Hermion* goes one step further and also presents runtime type information or dynamic collaborations

between various types of artifacts, while fluid source code views just focus on methods.

NavTracks [SING 05] keeps track of the navigation history of software developers. Using this history, NavTracks forms associations between related source files (such as class files) and can hence presents related entities to the developers. *Hermion* also shows related artifacts (for instance, in its reference view), but exploits dynamic information to extract the relations between artifacts. Thus, relatedness of artifacts is interpreted differently by *Hermion*; its aim is to improve the understanding of static source code and not recommend artifacts developers might want to browse next as NavTracks does.

Mylyn [KERS 05, KERS 06] monitors the programmer's activity in the IDE to get a degree-of-interest model for program elements scattered across a large code base, so the IDE can reveal code elements that are likely to be important for the task at hand [KERS 05]. Thus, Mylyn also presents related artifacts to the developer. *Hermion*, however, does not claim to identify task-relevant artifacts, but to link together artifacts used at runtime to boost the understanding of static code and identify collaboration not easily visible in this static code.

7.7 Summary of the Chapter

In this chapter we addressed the restrictions and limitations current IDEs impose on developers faced with the task of understanding an object-oriented system, in particular if implemented in a dynamically-typed language. We motivated our work by addressing the questions:

- *How do we integrate dynamic information into an IDE's browsing and navigating mechanisms?* We identified various types of dynamic information useful for software maintenance and explored how such information can be integrated into the workflows and views of an IDE to support developers to gain an understanding of a large software space through enriched views and precise navigation.
- *How can we efficiently collect dynamic data of a running application in the IDE?* We introduced an approach based on partial behavioral reflection to efficiently and selectively collect dynamic data at different levels of granularity (including sub-method data about variable accesses) at runtime.

We implemented an experimental IDE called *Hermion* which integrates dynamic information, validated its approach by applying it to two real-world applications, and measured the efficiency of the dynamic data gathering using these applications. Moreover, we observed developers experimenting with *Hermion* to assess its benefit on navigation and understanding of a software system.

While *Hermion* focuses on improving the understanding of imprecise static source code and the execution flow in methods (for instance, which if statements are executed), its support for explicitly representing dynamic collaboration is limited. In particular higher level collaboration, for instance which packages communicate with each other, is not represented in *Hermion*. Moreover, collaboration patterns are not visualized, thus dynamic communication is rather hard to understand with *Hermion*'s reference views which just list all classes referenced by an artifact. Additionally, *Hermion* does not represent software features in the IDE. We present in the following chapters proposals addressing these limitations of *Hermion*. The next chapter reports on *Senseo* which aims at also representing higher level collaboration patterns between packages, classes, and methods. *Senseo* also adapts the integration of dynamic information to the needs of a statically-typed language. Chapter 9 introduces visualizations of dynamic collaboration patterns to ease their understanding while Chapter 10 presents a means to explicitly represent software features in an IDE.

Chapter 8

Senseo – High Level Augmentations of IDEs with Dynamic Information

8.1 Introduction

8.1.1 Positioning *Senseo*

This chapter reports on *Senseo*, an enhancement of the Eclipse Java IDE. *Senseo* augments the static source perspectives with dynamic information and primarily aims at tackling the problem of not having a representation of dynamic collaboration between distant but conceptually related source elements in IDEs. Additionally, *Senseo* contributes to a better understanding of static source code and execution flow in and between methods. Moreover, *Senseo* helps developers to spot performance bottlenecks as it visually highlights in Eclipse's source views such as the package explorer dynamic metrics like the number of objects created in a method or class. In this way, *Senseo* also provides limited support for quality assessment regarding performance issues. By providing a view on the collaboration patterns between packages or classes, *Senseo* also contributes to a better overview of the system. Figure 8.1 subsumes all IDE problems addressed by *Senseo*. As illustrated by this figure, *Senseo* is able to improve the fulfillment of all identified development activities, at least to some degree.

Senseo is tailored to the needs of Java, a statically-typed language, and embeds the dynamic information in the Eclipse IDE. In contrast to *Hermion*, the focus of *Senseo* is rather on higher level artifacts than sub-method elements. *Senseo* primarily represents dynamic collaboration between source artifacts such as packages, classes, or methods. In a dedicated collaboration view embedded in Eclipse, *Senseo* displays for the currently selected artifact all other elements using or used by this particular artifact (callers or callees). To embed dynamic information such as argument or receiver types of message sends in the source code views, *Senseo* enhances the tooltips of Eclipse.

	Activity / Problem	Feature investigation	Feature implement-ation	Artifact investigation	Dependency investigation	Runtime interaction investigation	Artifact usage investigation	Execution pattern investigation	Quality assessment	Domain concept analysis
Over-loaded views	No overview	✔	✔		✔				✔	
I	Quality assessment support poor						(✔)		(✔)	(✔)
Static views	Distributed artifacts	✔		✔	✔	✔		✔	✔	✔
	Collaboration hidden	✔		✔	✔	✔		✔	✔	✔
	Execution paths hidden			✔	✔	✔	✔	✔		
	Imprecise static source code			✔		✔		✔		

Figure 8.1: *Senseo* contributes to a better system overview, makes visible dynamic collaboration between distant artifacts, improves the understanding of static source code and execution flow in and between source artifacts, and even offers limited support for quality assessment.

We discuss *Senseo* in the remainder of the chapter. We first introduce the approach and motivate the need to embed dynamic information in the IDE on the basis of a use case. Second, we describe how to gather dynamic information about Java systems and how we integrate and visualize this information. We validate *Senseo* by means of a controlled empirical experiment with 30 professional Java developers and a thorough evaluation of the performance of our approach to collect runtime information.

8.1.2 Introduction to *Senseo*

Maintaining object-oriented systems is complicated by the fact that conceptually related code is often scattered over a large source space. The use of inheritance, interface types, and polymorphism leads to code that is hard to understand, as it is unclear which concrete methods are invoked at runtime at a polymorphic call site even in statically-typed languages. As integrated development environments (IDEs) typically focus on browsing static source code, they provide little help to reveal the execution

paths a system actually takes at runtime. Being able to reconstruct the execution flow of a system while working in the IDE can, however, lead to better program understanding and to more focused navigation in the source code. In this chapter we show that developers can more efficiently maintain object-oriented code in the IDE if the static views of the IDE are augmented with dynamic information.

The importance of execution path information becomes clear when inspecting Java applications employing abstract classes or interfaces. The source code of such applications usually refers to these abstract types, while at runtime concrete subtypes are used. However, if the source code just refers to abstract types, it can be difficult to identify the concrete classes actually used at runtime, since there may exist a large number of concrete implementations. Similarly, when examining source code that invokes a particular method, a large list of candidate method implementations may be generated. Static analysis alone will not tell you how frequently, if at all, each of these candidates is actually invoked. Such information is nevertheless crucial to assess the performance impact of particular code statements.

Developers usually resort to debuggers to determine the actual execution flow of an application. Unfortunately, information extracted during a debugging session is volatile, that is, it disappears at the end of the session. Furthermore, such information is bound to a specific execution; in general, it cannot be used to tell which runtime types occur how often at a specific place in source code. To analyze and improve the performance of a system, developers typically use profilers, which suffer from similar drawbacks as debuggers: the collected dynamic information is not integrated in the static source views in the IDE; developers use such tools only occasionally instead of continuously benefiting from dynamic information that is directly available in the static source views.

We present an approach to dynamically analyze systems and to augment the static views of IDEs with dynamic information. We implemented this approach in *Senseo*, an Eclipse plugin that enables developers to dynamically analyze Java applications. *Senseo* enriches the source views of Eclipse with several kinds of dynamic information such as presenting which concrete methods a particular method invokes how often at runtime, which methods invoke this particular method, and how many objects or how much memory is allocated in particular methods. The gathered information is aggregated over several runs of the subject system; the developer decides which runs to take into account. *Senseo* also contributes two other means to integrate dynamic information: first, a view on the dynamic collaborations between different source artifacts, which illustrates communication at the level of packages, classes and methods, and,

second, the Calling Context Ring Chart (*CCRC*) [MORE 09], a navigable visualization of the system's Calling Context Tree (CCT) [AMMO 97].

To validate the usefulness of *Senseo* in practice, we conducted a controlled user experiment with 30 professional Java developers to obtain reliable quantitative and qualitative feedback about the impact on developer productivity contributed by *Senseo* and the dynamic information it integrates in Eclipse. The subjects solved five typical software maintenance tasks in an unfamiliar, medium-sized software system. While half the subjects only used the standard Eclipse IDE, the other half additionally used the *Senseo* plugin. The experiment shows that the availability of dynamic information as provided by *Senseo* yields a significant decrease in time of 17.5% and a significant increase in correctness of 33.5%.

We contribute (i) an approach to gather and integrate dynamic information in the Eclipse IDE, (ii) a controlled user experiment to validate the practical usefulness of the approach, and (iii) a detailed performance evaluation of *Senseo*, our implementation of the approach. With respect to our prior work [RÖTH 09d, RÖTH 09e], (ii) and (iii) are novel, original contributions.

The chapter is structured as follows: In Section 8.2 we present a use case motivating the need for dynamic information within the IDE. Section 8.3 introduces *Senseo*, a plugin that integrates dynamic information in the Eclipse IDE. Section 8.4 explains our approach to gather dynamic information from a running application. Section 8.5 validates the practical usefulness of *Senseo* for software maintenance tasks with a controlled experiment involving 30 professional developers. Section 8.6 reports on the efficiency of *Senseo*. Section 8.7 presents related work. Finally, Section 8.8 summarizes the chapter.

8.2 Motivation

Senseo aims at improving understanding and maintenance of object-oriented software systems by providing the developer dynamic information collected from multiple runs of an application, such as from the execution of unit tests. In order to motivate the need for exposing dynamic information in the IDE, we consider the Eclipse JDT[1], a set of plug-ins implementing the Eclipse Java IDE. JDT encompasses interfaces and classes modeling Java source code artifacts, such as classes, methods, fields, or local variables. Clients of this representation usually refer to interface

[1] http://www.eclipse.org/jdt

types, such as `IJavaElement` or `IJavaProject`, as the following code snippet found in `JavadocHover` illustrates:

```
IJavaProject javaProject = null;
IJavaElement element = elements[0];
if (element.getElementType() == IJavaElement.FIELD){
   javaProject = element.getJavaProject();
} else if (element.getElementType() == IJavaElement.
   LOCAL_VARIABLE) {
   javaProject = element.getParent().getJavaProject()
   ;
}
```

This code is difficult to understand due to the lack of information about runtime types of variables and any other dynamic information: (i) it is unclear which `getJavaProject` methods are invoked at runtime; (ii) the variable `javaProject` could still be `null` at the end of the code snippet, as not all possible types of elements might be covered by the conditionals; (iii) the execution frequency of this code and thus its performance impact is unknown.

These questions cannot be easily answered using only the IDE's static source views because there are more than ten different implementations of the method `getJavaProject` in the JDT, thus, we do not know which implementations are actually used. Furthermore, JDT contains many interfaces and classes implementing `IJavaElement`, therefore, we cannot statically determine which types of elements are used at runtime in this code.

Using a debugger, we find out that `element` is of type `SourceField` in one scenario. However, we know that debuggers focus on specific runs, thus we still cannot know all the different types `element` has in this code. To reveal all types of `element` and all `getJavaProject` methods invoked by this polymorphic call site, we would have to debug many more scenarios, which is very time-consuming as this code is executed many times for each system run.

For all these reasons, it is much more convenient for a developer if the IDE itself could show dynamic information aggregated over several runs within the static source views, that is, Eclipse's source code viewer should show precisely which methods are invoked at runtime, including detailed runtime types for receiver, arguments, and return values. In addition, information about the number of method invocations or object allocations helps developers identify performance bottlenecks in an application. If developers are interested in a specific execution, *Senseo* also allows them to just analyze the information from this single scenario. If

source code enriched with dynamic information changes, the recorded dynamic information about this piece of code is invalidated.

8.3 Integrating Dynamic Information in IDEs

In this section we present our approach to augment IDEs with dynamic information, towards the goal of supporting the understanding of runtime behavior of applications. First, we present the architecture of *Senseo*, an Eclipse plugin implementing our approach. Second, we discuss different kinds of dynamic information that can support program understanding. Third, we illustrate how *Senseo* integrates and visualizes such dynamic information within Eclipse.

8.3.1 Architecture

Dynamic information can be collected using a modified Java Virtual Machine (JVM), with a profiling agent in native code using the standard JVM Tool Interface (JVMTI), or with the aid of program transformation or bytecode instrumentation techniques. For portability and compatibility reasons, we chose the last approach. Instead of using a low-level bytecode engineering library to instrument code, we use high-level aspect-oriented programming (AOP) [KICZ 97] to specify instrumentation as an aspect. As we discovered in Section 2.2.2, AOP has several advantages compared to other approaches such as a low data gathering overhead. It also supports precise selection of the artifacts to be analyzed and ensures ease of maintenance and extension of the concrete specifications of the information to be collected.

MAJOR [VILL 08, VILL 09], an aspect weaver that supports comprehensive aspect weaving into every class linked in a JVM, including the standard Java class library, vendor specific classes, and dynamically loaded or generated classes, was identified in Section 2.2.2 as an appropriate technique to gather dynamic information to be integrated with *Senseo* into Eclipse. A main advantage of *MAJOR* is also its ability to cover the complete program execution, which clearly separates this technique from other, similar tools such as *J [DUFO 03a] or JFluid [DMIT 04b] (cf. Section 2.2.2). *MAJOR* is based on the standard AspectJ [KICZ 01] compiler and weaver and uses advanced bytecode instrumentation techniques to ensure portability [BIND 07]. The instrumentation code *MAJOR* has woven is executed immediately after the JVM bootstrapping.

The application to be analyzed is executed in a separate application JVM where *MAJOR* weaves the data-gathering aspect into every loaded

class, while the Eclipse IDE with the *Senseo* plugin runs in a standard JVM to avoid perturbations. While the subject system is still running, the gathered dynamic data is periodically transmitted from the application JVM to Eclipse using a socket. We do not have to halt the application to obtain its dynamic data. *Senseo* receives the transfered data, processes it, and stores the aggregated information in its own storage system which is optimized for fast access from the IDE (see Figure 8.2).

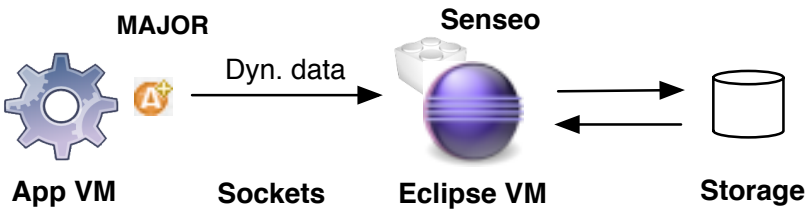

Figure 8.2: Setup to gather dynamic information.

To analyze the application dynamically within the IDE, developers have to execute it with *Senseo*. Before starting the application, developers can define what kind of dynamic information should be gathered at runtime. By default, all packages and classes of the application and the Java class library are dynamically analyzed. However, developers can restrict the analysis to specific classes or even methods to reduce the analysis overhead if only specific areas need to be observed. *Senseo* aggregates dynamic information over all application runs executed with it, but developers can clear the store and start afresh.

8.3.2 Dynamic Information

The *Senseo* plugin integrates the following dynamic information into the IDE.

Method invocation We extract the following information:

Invoked Methods Often, invoked methods are not implemented in the declared type of a receiver, but in a super- or subtype, if inheritance or dynamic binding are used. Providing information in the IDE about the methods invoked helps developers better understand collaborations between objects and the runtime execution flow.

Developers can also ask to gather further detailed information about method invocation:

- *Receiver types.* Often subtypes of the type implementing the method invoke the method at runtime. Knowing receiver types and their frequency can further increase program understanding.
- *Argument types.* Information about actual argument types and their frequency increases the understanding for a method, *i.e.* how it is used at runtime.
- *Return types.* Knowing the concrete types of return values and their frequency helps developers better understand communication between different methods.

Number of invocations. This dynamic metric helps developers quickly identify hot spots in code, that is, very frequently invoked methods and classes containing such methods. Furthermore, methods never invoked at runtime become visible, which is useful when removing dead code or extending the test coverage of the application's test suite.

Number of created objects. A developer usually cannot tell by reading source code alone how many objects are created at runtime in a class, a method or a line of code. This dynamic metric can help one to locate inefficient code that creates many objects.

Allocated memory. Objects vary in size. Many small objects might not pose an issue, whereas creating large objects could result in high memory consumption. Hence, we also provide a dynamic metric recording memory allocation of various source artifacts, such as classes or methods. This metric can be combined with the number of created objects metric to reveal which types of objects consume most memory and thus are candidates for optimization.

CCT. The CCT [AMMO 97] allows dynamic information to be collected separately for each calling context. A calling context is a stack of methods that have been invoked but have not yet completed. The CCT helps the dynamic inter-procedural control flow of an application to be analyzed. Figure 8.3 illustrates a code snippet together with the corresponding CCT (showing only method invocation counts as metric).

8.3.3 Enhancements to the IDE

We now describe how these different kinds of dynamic information are presented in Eclipse by *Senseo*.

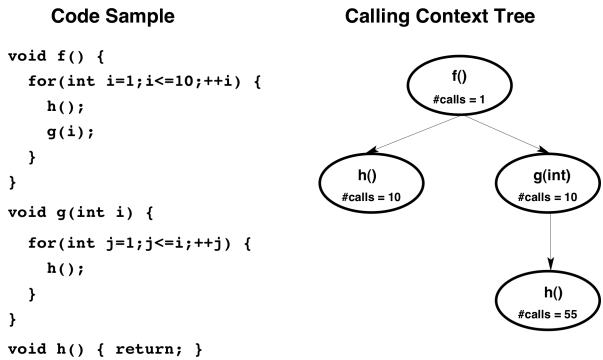

Figure 8.3: Sample code and its corresponding CCT.

Source code enhancements. We use *tooltips*, small windows that pop up when the mouse hovers over a source element to complement source code without impeding its readability. *Senseo* tooltips are interactive; that is, the developer can open the class of a receiver type by clicking on it.

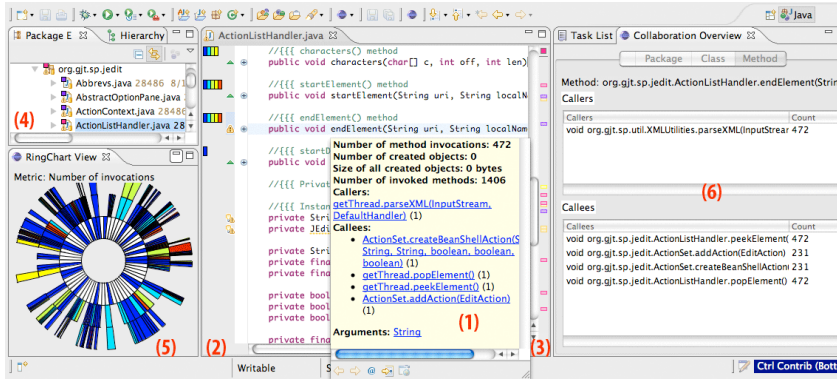

Figure 8.4: All six interactive views of *Senseo*.

Method header tooltip When the mouse hovers over the method name in a method header, the tooltip shows (i) all callers invoking that particular method, (ii) all callees of the method, and, optionally, (iii) all argument and return value types. For each piece of information we also show how often a particular invocation occurred. Figure 8.4 (1) shows a tooltip for the concrete method `endElement`. If available, we also show information about argument and return types when the mouse is over the declared arguments of a method or the declared return type. These tooltips also

display how often specific argument and return value types occurred at runtime.

Method body tooltip Source elements in the method body also support tooltips. For each call site of the method, we provide the dynamic callee information as for the method name, namely concretely invoked methods, optionally along with argument or return types that occurred in this method for that particular method invocation at runtime. This information is always accompanied with the number of occurrences and the relative frequency of the specific types at runtime.

Ruler columns. In Eclipse, the source code editor comes with two ruler columns: The left one shows local annotations (errors, warnings, etc.) while the right one presents an overview of all annotations from the entire document. We extended these two rulers to also display dynamic information. For every executed method in a Java source file, the overview ruler (Figure 8.4 (3)) presents how often it has been executed on average per system run using three different icons colored using a heat scheme: *blue* means only a few, *yellow* several, and *red* many invocations [RÖTH 09c]. Clicking on such an annotation icon triggers a jump to the declaration of the method in the file. The ruler on the left (Figure 8.4 (2)) shows the frequency of invocation of a particular method on a scale from 1 to 6, compared to all other invoked methods. A completely filled bar for a method denotes methods that have been invoked the most in this application. These two rulers allow developers to quickly identify hot spots in their code.

To associate the continuous distribution of metric values to a discrete scale with three or more representations (*e.g.* red, yellow, and blue), we use the *k-means* clustering algorithm [LLOY 82].

The two rulers are also enriched with tooltips showing more fine-grained dynamic information. Hovering over a heat bar in the left column or over the annotation icon in the right bar triggers a tooltip displaying precise values, such as exact total numbers of invocations or even the number of invocations from specific methods or receiver types.

Developers can choose between different kinds of dynamic information to be visualized in the rulers, such as the number of objects a method creates or the amount of memory it allocates, either on average or in total over all executions. Such metrics allow developers to quickly assess the runtime complexity of specific methods and thus to locate candidate methods for optimization. The dynamic information to be displayed is set in the Eclipse preferences.

Package Explorer. The package explorer is the main tool in Eclipse used to locate packages and classes of an application. *Senseo* augments the package explorer with dynamic information to guide the developer at a high level to the source artifacts of interest, see Figure 8.4 (4). For this purpose, we annotate packages and classes in the explorer tree with icons denoting the degree to which they contribute to the selected dynamic metric such as amount of allocated memory. Thus a class aggregates the metric value of all its methods, a package the value of all its classes. Similar to the overview ruler the metric values are mapped to *blue*, *yellow*, and *red* package explorer icons representing a heat coloring scheme [RÖTH 09c].

Collaboration View. In a separate view next to the source code editor (Figure 8.4 (6)), *Senseo* presents all dynamic collaborators for the currently selected artifact. For instance, if a method has been selected, the collaboration view shows all packages or classes invoking methods of the package or class in which the selected method is declared (callers). The collaboration view also shows all packages or classes with which the package or class declaring the method is actively communicating (callees). For the method itself, the collaboration view lists all direct callers and callees.

Calling Context Ring Chart (*CCRC*). The *CCRC* [MORE 09] offers a compact visualization of a CCT and provides navigation mechanisms to locate and explore subtrees of interest for the software maintenance task at hand (Figure 8.4 (5)). Like the Sunburst visualization [STAS 00], *CCRC* uses a circular layout. The CCT root is represented as a circle in the center. Callee methods are represented by ring segments surrounding the caller's ring segment. For a detailed analysis of certain calling contexts, CCT subtrees can be visualized separately and the number of displayed tree layers can be limited.

8.4 Collecting Dynamic Information

In this section we explain our approach to collecting dynamic information using AOP.

Senseo requires flexible support to aggregate dynamic information. For instance, runtime type information is needed separately for each pair of caller and callee methods, while memory allocation metrics need to be aggregated for the whole execution of a method (including direct and indirect callees). In order to support different ways of aggregating metrics,

a data structure is needed to store dynamic information separately for each executed calling context. The CCT [AMMO 97] perfectly fits this requirement.

Our CCT representation is designed for extensibility so that additional metrics can be easily integrated. Each CCT node stores dynamic information and refers to an identifier of the target method for which the metrics have been collected. It also links to the parent and child nodes for navigation in the CCT.

Our implementation leverages *MAJOR* [VILL 08, VILL 09], an aspect weaver with two distinguishing features. First, *MAJOR* supports complete method coverage. Method invocations through reflection and callbacks from native code into bytecode are correctly handled. Second, *MAJOR* provides efficient access to complete calling context information through customizable, thread-local shadow stacks. Using the pseudo-variables `thisStack` and `thisSP`, the aspect gets access to the array holding the current thread's shadow stack, respectively to the array index (shadow stack pointer) corresponding to the currently executing method.

Figure 8.5 illustrates three advices[2] of our aspect for CCT construction and dynamic information collection. In the `CCTAspect`, each thread generates a separate, thread-local CCT. The shadow stack is an array of `CCTNode` instances, representing nodes in the thread-local CCT. A special root node is stored at position zero. Periodically, after a configurable number of profiled method calls, each thread integrates its thread-local CCT into a shared CCT in a synchronized manner. This approach reduces contention on the shared CCT, yielding significant overhead reduction in comparison with an alternative solution where all threads directly update a shared CCT upon each method invocation.

The first advice in Figure 8.5 intercepts method entries and pushes the `CCTNode` representing the invoked method onto the shadow stack. To this end, it gets the caller's `CCTNode` instance from the shadow stack (i.e., at position `sp-1`) and invokes the `profileCall` method, which takes as argument an identifier of the callee method. We use static join points, accessed through AspectJ's `thisJoinPointStaticPart` pseudo-variable, to uniquely identify method entries; they provide information about the method signature, modifiers, etc. The `profileCall` method returns the callee's `CCTNode` instance and increments its invocation counter; if the same callee has not been invoked in the same calling context before, a new `CCTNode` instance is created as child of the caller's node.

[2] Aspects specify *pointcuts* to intercept selected *join points* in the execution of programs, such as method calls. *Advices* adapt join points with code to be executed before, after or around them.

```
aspect CCTAspect {
 before(): execution(* *(..)) {
   CCTNode[] ss = thisStack; int sp = thisSP;
   ss[sp] = ss[sp-1].
            profileCall(thisJoinPointStaticPart);
   ss[sp].storeRcvArgsRuntimeTypes(thisJoinPoint);
 }

 after() returning(Object o): execution(* *(..)) {
   CCTNode[] ss = thisStack; int sp = thisSP;
   ss[sp].storeRetRuntimeType(o);
   ss[sp] = null;
 }

 after() returning(Object o): call(*.new(..)) {
   CCTNode[] ss = thisStack; int sp = thisSP;
   ss[sp].storeObjAlloc(o);
 }
 ...
}
```

Figure 8.5: Simplified excerpt of the CCTAspect

The second advice in Figure 8.5 deals with normal method completion, popping the method's entry from the shadow stack. For simplicity, here we do not show cleanup of the shadow stack in the case of a method completing abnormally by throwing an exception [VILL 09].

The third advice intercepts object creation to keep track of the number of created objects and the memory allocated for each calling context. The method `storeObjAlloc(Object)` uses the object size estimation functionality of the `java.lang.instrument` API to update the memory allocation statistics in the corresponding `CCTNode` instance.

`CCTAspect` collects the receiver, argument and result runtime types using dynamic join points.

During execution, the aspect code periodically sends the collected metrics to the *Senseo* plugin in the IDE. Upon metrics transmission, thread-local CCTs of terminated threads are first integrated into the shared CCT. Afterwards, the shared CCT is traversed to aggregate the metrics as required by *Senseo*. Finally, the aggregated metrics are sent to the plugin through a socket. Metrics aggregation and serialization may proceed in parallel with the program threads, since they operate on thread-local CCTs most of the time.

8.5 Validation

We conducted a controlled experiment with 30 professional Java developers to evaluate the benefits of *Senseo* [RÖTH 09d] for software maintenance. We now describe the experimental design, the subjects, the evaluation procedure, the final results (including qualitative feedback), as well as threats to validity.

8.5.1 Experimental Design

This experiment aims at quantitatively evaluating the impact of *Senseo* and the dynamic information it integrates in the Eclipse IDE on developer productivity in terms of efficiently and correctly solving typical software maintenance tasks. We therefore analyze two variables in this experiment: *time spent* and *correctness*. This experiment also reveals which kind of tasks benefit the most from the availability of dynamic information in the IDE. The experimental design we opted for is similar to the one applied in the study of Cornelissen *et al.* [CORN 09] which evaluated a trace visualization tool called *EXTRAVIS*.

Study Hypotheses. We claim that the availability of *Senseo* reduces the amount of time it takes to solve software maintenance tasks and that it increases the correctness of the solutions. Accordingly, we formulate the following two null hypotheses:

- $H1_0$: Having *Senseo* available does not impact the time for solving the maintenance tasks.
- $H2_0$: Having *Senseo* available does not impact the correctness of the task solutions.

Consequently, we formulate these two alternative hypotheses:

- H1: Having *Senseo* available reduces the time for solving the maintenance tasks.
- H2: Having *Senseo* available increases the correctness of the task solutions.

We test the two null hypotheses by assigning each subject to either a control group or an experimental group. While the experimental group has *Senseo* available for answering typical software maintenance tasks and questions, the control group uses a standard Eclipse IDE; otherwise

Table 8.1: Average expertise in control and experimental group.

Expertise variable	*Control group*	*Exper. group*
Years of experience	4.73	4.40
Java experience [0..4]	2.93	2.80
Eclipse experience [0..4]	2.80	2.67
Unfamiliar code exp. [0..4]	2.73	2.73

there is no difference in treatment between the two subject groups. As both groups have nearly equal expertise, differences in time or solution correctness can be attributed to the availability of the *Senseo* plugin.

Study Participants. We asked 30 software developers working in industry (24) or with former industrial experience (6) to participate in our experiment. Participation was voluntary and unpaid. All subjects answered a questionnaire asking for their expertise with Java, Eclipse, and specific skills in software engineering, such as how often they work with unfamiliar code. All participants are familiar with Java and the Eclipse IDE.

The subjects have between one and 25 years of professional experience as software engineers (average 4.8 years, median 4 years). 27 subjects have a university degree in computer science while three subjects either studied in another area or learned software engineering on the job. The subjects are very heterogeneous and thus fairly representative (seven different nationalities, working for eight different companies). In a Likert scale [LIKE 32] from 0 (no experience) to 4 (expert) subjects rated themselves on average 2.93 for Java experience, 2.73 for Eclipse experience, and 2.72 for experience in working with unfamiliar code. All these ratings correspond to "very experienced".

To assign the 30 subjects to either the experimental or the control group, we used the obtained expertise information. To assess the expertise we considered four variables as given by the subjects: number of years of professional experience in software engineering, experience with Java, Eclipse and with maintaining unfamiliar code. For each subject we searched for a pair with similar expertise concerning these variables and then randomly assigned these two persons to either of the two groups. This leads to a very similar overall expertise in both groups as shown in Table 8.1.

Subject System and Tasks. As a subject system we have chosen *jEdit*[3], an open-source text editor written in Java. JEdit consists of 32 packages with 5275 methods in 892 classes totaling more than 100 KLOC. We opted for jEdit as a subject system as it is medium-sized and representative of many software projects found in industry. JEdit has a long history of development spanning nearly ten years and involving more than ten developers. Even though it has been refactored several times, a careful analysis of the code quality revealed several design flaws, such as the use of deprecated code, tight coupling of many source entities to package-external artifacts, and lack of cohesion in almost all packages, all of which makes jEdit hard to understand. We expect many industrial systems to have similar quality problems, thus we consider jEdit to be a well-suited subject application fairly typical for many industrial systems developers come across on their job. Furthermore, the domain of a text editor is familiar to everyone, thus no special domain-knowledge is required to understand jEdit.

The tasks we gave the subjects are concerned with analyzing and gaining an understanding for various features of jEdit. While choosing the tasks, our main goal was to select tasks representative for real maintenance scenarios. Furthermore, these tasks must not be biased towards dynamic analysis. To assure that these criteria are met we selected the tasks according to the framework proposed by Pacione *et al.* [PACI 04]. They identified nine principal activities for reverse engineering and software maintenance tasks covering both static and dynamic analysis. Based on these activities they propose several characteristical tasks including all identified activities. We thus design our tasks following this framework to respect all nine principal activities, which avoids a potential bias towards *Senseo*.

This leads us to the definition of five tasks, each divided into two subtasks, resulting in ten different questions we asked to the subjects. Table 8.2 outlines all five tasks and their subtasks and explains which of Pacione's activities they cover. Task five is special since we use it as a "time sink task" to avoid ceiling effects [ARIS 07]. Subjects that can answer the questions quickly might spend considerably more time on the last task when they notice that there is still much time available, so the addition of a time-consuming task at the end which is not considered in the evaluation ensures that subjects have a constant time pressure for all relevant tasks. The first four tasks still cover all of Pacione's activities.

All questions are open, that is, subjects cannot select from multiple choices but have to write a text in their own words. Beforehand, the

[3]http://www.jedit.org/

Table 8.2: The five software maintenance tasks.

Task	*Activities*	*Description*
1.1	A 1, 9	Locating a feature in code and naming the packages and architectural layers in which it is implemented
1.2	A 1, 4 ,5	Describing package collaborations in this feature
2.1	A 8	Comparing fan-in, fan-out of three classes
2.2	A 4, 5, 6, 8	Describing coupling between the packages of these three classes
3.1	A 1, 3, 4, 5	Analyzing the order in which methods of a class are invoked
3.2	A 1, 3, 5, 7	Locating clients of this class and analyzing the communication patterns between the class and its clients
4.1	A 4, 5, 8, 9	Comparing two features on a fine-grained method level to locate a defect in a feature
4.2	A 2	Correcting this defect by comparing it to the other, flawless feature
5.1	A 4, 5, 6, 7	Exploring an algorithm in a specific class and analyzing its performance
5.2	A 5, 6, 7, 8	Comparing this algorithm to another, similar algorithm in terms of efficiency

experimenters solved all tasks themselves to prepare model answers according to which the subjects' answers were corrected.

Experimental Procedure. We gave the subjects a short five minute introduction to the experiment setup. Subjects from the experimental group additionally received a 20 minute introduction to *Senseo*, following a prepared script to ensure that every subject receives the same information. We provided the *Senseo* subjects with a short description and a screenshot highlighting and explaining the core features of *Senseo*, to serve as a reference during the experiment.

Afterwards, we started the experiment. We supervised all subjects during the entire experiment and recorded the time they took to answer each question. Concerning infrastructure, each subject obtained the same pre-configured Eclipse installation we distributed in a virtual image. The only difference between the control group and the experimental group was the availability of the *Senseo* plugin, otherwise the Eclipse IDE was configured in exactly the same way.

We provided the *Senseo* group with pre-recorded dynamic information obtained by executing all actions from the menu bar of jEdit to make sure that the pre-recorded information is not biased towards the experiment tasks. We provided pre-recorded dynamic information to control the variable of tracing the appropriate software features. Although it does not take much time to gather dynamic information with *Senseo*, freeing subjects from this task makes sure that the subjects' performance in the experiment is only dependent on how *Senseo* presents the information and not on which information has been recorded. As the control group did not receive any dynamic information, we clearly stated in the task descriptions how to run and analyze the feature under study with the conventional debugger in Eclipse.

Variables and Evaluation. The two dependent variables we study in this experiment are *time* the subjects spend to answer the questions, and *correctness* of the answers. Keeping track of the answer time is straightforward as we prohibited going back to previously answered questions. We simply record the time span between the starting time of one question and the next. Correctness is measured using a score from 0 to 4 according to the overlap with the model answers, which forms a set of expected answer elements (usually the names of certain source artifacts).

The only independent variable in our experiment is whether the *Senseo* plugin is available in the Eclipse IDE to the subjects during the experiment.

We apply the parametric, one-tailed Student's t-test to test our two hypotheses at a confidence level of 95% (α=0.05). To validate that the t-test can be used, we first apply the Kolmogorov-Smirnov test to verify normal distribution and then Levene's test to check for equality of variance in the sample.

8.5.2 Results and Discussion

In this section we analyze the results obtained in the experiment. First, we evaluate the results for time and correctness. Second, we identify for which types of tasks the availability of dynamic information in the IDE is most useful. Finally, we evaluate the qualitative feedback we gathered by means of a debriefing questionnaire.

Only three subjects could not complete the time sink task (task 5) in the two hours we allotted, but everybody finished the four relevant tasks.

Table 8.3: Statistical evaluation of the experimental results.

Group	*Mean*	*Stdev.*	*K.-S.*	*Lev F*	*t*	*p*
Time [m]:						
Eclipse	114.80	20.62	0.27			
Senseo	94.73 (-17.5%)	12.4	0.18	3.06	3.23	.0016
Correctness (points):						
Eclipse	11.33	2.58	0.31			
Senseo	15.13 (+33.5%)	2.10	0.24	0.22	4.42	.0001

Time. On average, the *Senseo* group spent 17.5% less time solving the maintenance tasks. The time spent by the two groups is visualized as a box plot in Figure 8.6.

To statistically verify whether *Senseo* has an impact on the time to answer the questions, we test the null hypothesis $H1_0$ which says that there is no impact. We successfully applied the Kolmogorov-Smirnov and the Levene test on the time data (see Table 8.3), thus we are able to apply Student's t-test to evaluate $H1_0$. The application of the t-test allows us to reject the null hypothesis and instead accept the alternative hypothesis, which means that the time spent is statistically significantly reduced by the availability of *Senseo* as the p-value is with 0.0016 considerably lower than α=0.05 (see Table 8.3).

From the observations of subjects during the experiment, from their informal feedback during the debriefing interviews, and particularly from the formal questionnaires (see below), we could conclude that subjects using *Senseo* were more efficient due to the following reasons: (i) the availability of dynamic information in the source code tooltips helps developers to more quickly gain an understanding how source artifacts communicate with each other, (ii) the visualizations of dynamic information such as number of method invocations shown in ruler columns and package tree enable developers to quickly spot which source elements are executed and how often, and (iii) as the collaboration view accurately presents all source artifacts that are related or collaborate with a selected source entity such as a package, class or method, developers can more quickly navigate to code relevant for a specific task.

Correctness. The *Senseo* group's answers for the four maintenance question are 33.5% more correct, which is also shown in the box plot in Figure 8.6.

To test the null hypothesis $H2_0$, which suggests that there is no effect of the availability of *Senseo* on answer correctness, we can also use the Stu-

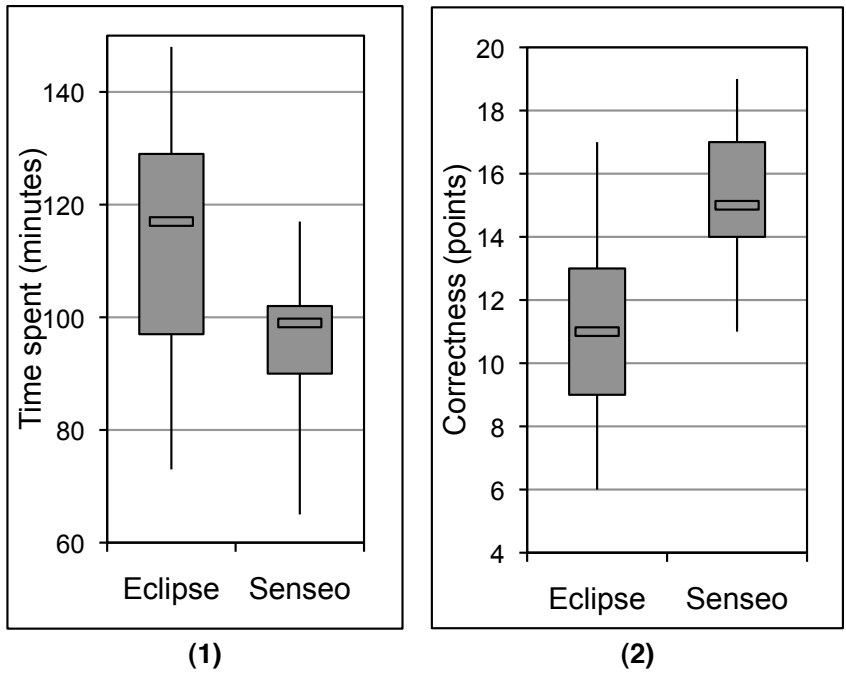

Figure 8.6: Box plots comparing time spent and correctness between control and experimental group.

dent's t-test as the Kolmogorov-Smirnov and the Levene test succeeded for the correctness data (compare Table 8.3). As the t-test gives a p-value of 0.0001 which is clearly below α=0.05, we reject the null hypotheses and accept the alternative hypothesis H2, which means that having *Senseo* available during software maintenance activities helps developers to more correctly solve maintenance tasks.

The evaluation of the questionnaire, the observations during and the informal interviews after the experiment allowed us to attribute the improvements in correctness to the same techniques of *Senseo* that also improved the efficiency: (i) precise information about runtime collaboration or execution paths as highlighted in the extended source tooltips enables developers to accurately navigate to dependent artifacts, (ii) information about execution complexity (number of method calls or number and size of created objects shown in ruler columns or package tree) eases the correct identification of inefficient code, and (iii) accurate overviews of collaborating artifacts given by the collaboration view supports developers in exploring all relevant parts of the system to completely address a task.

Table 8.4: Task individual performance concerning time required and correctness.

Task	Time [m]		Correctness (points)	
	Eclipse	*Senseo*	*Eclipse*	*Senseo*
Task 1	511	425 (-16.8%)	38	53 (+39.5%)
Task 2	388	340 (-12.4%)	58	79 (+36.2%)
Task 3	437	291 (-33.4%)	52	69 (+32.7%)
Task 4	386	365 (-5.4%)	22	26 (+18.2%)

Table 8.5: Percentage of subjects using specific dynamic information in particular tasks.

Dynamic Information	*Task 1*	*Task 2*	*Task 3*	*Task 4*
Runtime types (Tooltip)	33%	47%	47%	20%
Number of invocations	53%	67%	40%	27%
Number of created objects	33%	47%	27%	13%
Number of exec. bytecodes	27%	33%	20%	7%
CCRC	7%	7%	0%	0%
Dynamic collaborators (callers, callees)	53%	80%	73%	33%

Task-dependent Results. We also analyzed the two variables, time spent and correctness, for each task individually to reveal which kinds of tasks benefit most from dynamic information integrated in Eclipse. Table 8.4 presents the aggregated results for time spent and correctness for each subject group and each task individually. Tasks 1, 2 and 3 benefit significantly from the availability of *Senseo* both in terms of time required to solve them and the correctness of the solution. However, for task 4 the benefit of *Senseo* is less pronounced.

Qualitative Feedback. We also collected qualitative feedback using a questionnaire to evaluate the impact of particular parts of *Senseo* on specific kinds of maintenance tasks. This evaluation yields answers to the question which *Senseo* feature and which kind of dynamic information is actually relevant or useful in what kind of maintenance tasks.

In Table 8.5 we list for each task the percentage of subjects that used a specific kind of dynamic information integrated by *Senseo* ("Did you use dynamic information X in task Y?"), and Table 8.6 presents how useful subjects rated each *Senseo* technique on a Likert scale from 0 (useless) to 4 (very useful).

Table 8.6: Mean ratings of the subjects for each feature of *Senseo* .

Dynamic Information	*Mean rating [0..4]*
Tooltip showing runtime types	3.6
Ruler column incl. dynamic info	3.2
Overview ruler column incl. dyn. info	3.0
Package tree incl. dynamic info	2.4
CCRC	2.1
Collaboration view	3.7

From the evaluation, we draw the conclusion that there are basically three kinds of tasks whose solution process is very well supported by the availability of dynamic information in IDEs: (i) tasks requiring developers to understand how different source artifacts collaborate or depend on each other, (ii) tasks in which developers have to assess how often code is executed or how complex its execution is, and (iii) tasks that require the developer to understand which code is related to a given feature. This conclusion agrees with the quantitative results discussed earlier where we revealed that task 1 (feature and collaboration understanding), task 2 (quality assessment) and task 3 (control flow understanding) benefited most from the availability of *Senseo*, while for task 4 (low level defect correction) dynamic information was less useful.

From the results evaluating the different *Senseo* concepts (Table 8.6), we conclude that developers particularly benefit from the availability of the collaboration views and runtime type information in source code. Also considered useful are visualizations of dynamic information in the source code columns such as the presentation of number of invoked methods in a method or class. The aggregated dynamic information presented in the package tree are perceived as less useful by the developers, probably because it is not meaningful to study runtime complexity at a high package level. The subjects also could not benefit from the *CCRC* as this visualization serves the rather specialized task of performance optimization which has not been directly covered by the maintenance tasks of the experiment.

8.5.3 Threats to Validity

In this section we discuss several threats to validity concerning this experiment. We distinguish between (i) construct validity, that is, threats due to how we operationalized the time and correctness measures, (ii) internal validity, that is, threats due to inferences between treatment and

effect during the analysis, and (iii) external validity which refers to threats concerning the generalization of the experiment results.

Construct Validity. Due to the operationalization of the time and correctness variables, the results might not hold in real, non-experimental situations. For instance, subjects could have been more attentive than they would be in their daily job, or they could have been more anxious as they were observed and assumed that their performance was being evaluated. However, we consider this threat to be negligible as we made clear that subjects' performance is not evaluated. Furthermore, this threat is likely to affect both the control and the experimental group equally.

Internal Validity. Some threats to internal validity originate from the subjects. First, subjects might not have the required expertise to properly solve the maintenance tasks. This threat is largely eliminated by preliminary assessment of the subjects' expertise concerning their Java, Eclipse and software maintenance skills. Additionally, we required them to not have expert knowledge in developing *jEdit*. Second, the experimental group might have had more knowledge than the control group. This threat is mitigated by assigning the subjects in a randomized manner to the two groups in a way that both groups have nearly equal expertise (see Table 8.1).

Other threats to internal validity stem from the maintenance tasks we prepared. First, the tasks could have been too difficult or time-consuming to solve. This threat is refuted by the fact that nearly all subjects from both groups could solve all tasks in time (except two from the control group and one from the *Senseo* group). Moreover, each question was answered fully correctly by at least one person from each group. Additionally, we asked subjects in the questionnaire directly how they judged the time pressure and the difficulty. On average, the ratings were 2.8 for time pressure (representing "felt no time pressure") and 3.1 for average difficulty of all tasks (which means "appropriately difficult"). Second, the threat that we formulated tasks favoring *Senseo* is largely limited as we used Pacione's established framework [PACI 04] to find the tasks used in the experiment. Third, a threat for the correctness evaluation is that the experimenters might have favored *Senseo* while grading subjects' answers. By initially building an answer model according to which the subjects answers were graded, we mitigated this threat. For the obtained answers the experimenters gave points as pre-defined in the answer model which in turn has been formulated and validated by two persons individually.

External Validity. Generalizing the results of the experiment could be unjustified due to the selection of tasks, subjects, or the application used in the experiment. This threat is mitigated since we selected the maintenance tasks carefully to follow Pacione's framework [PACI 04] of representative maintenance tasks. We furthermore asked open questions to the subjects to better model industrial reality than would be possible with multiple choice questions.

As the subjects work for different companies and have a high variety of education profiles, the study participants should be fairly representative for professional software developers and thus not impose a threat to generalization.

In Section 8.5.1 we described several reasons why jEdit is representative for many industrial systems. Additionally, we asked subjects at the end of the experiment how comparable in terms of maintainability they consider *jEdit* to be to systems they daily work with. On average, they gave on a Likert scale from 0 (totally different) to 4 (very representative) a rating of 3.1, which refers to "many similarities". Hence we are confident to have found with *jEdit* a system representative for most industrial applications.

8.6 Performance

In order to validate that *Senseo* offers sufficient performance to cope with real-world workloads, we evaluated the different sources of overhead and analyzed the amount of transmitted data for the DaCapo benchmarks[4] [BLAC 06]. For our measurements, we use *MAJOR*[5] version 0.6 with AspectJ[6] version 1.6.5 and the SunJDK 1.6.0_13 Hotspot Server Virtual Machine. We execute the benchmarks on a quadcore machine running CentOS Enterprise Linux 5.3 (Intel Xeon, 2.4GHz, 16GB RAM).

Figure 8.7 shows the overhead for CCT creation, collection of dynamic information (including the number of method invocations, the number of object allocations, the estimated allocated bytes, and the runtime receiver, argument, and return value types), as well as serialization and data transmission to the Eclipse plugin, including processing of the received data by the plugin. In this measurement setting, each benchmark is executed 15 times and the median execution time is taken for computing the overhead. For each run of each benchmark, the CCT and the gathered dynamic information are serialized and transmitted once upon

[4] http://dacapobench.org/

[5] http://www.inf.usi.ch/projects/ferrari/

[6] http://www.eclipse.org/aspectj/

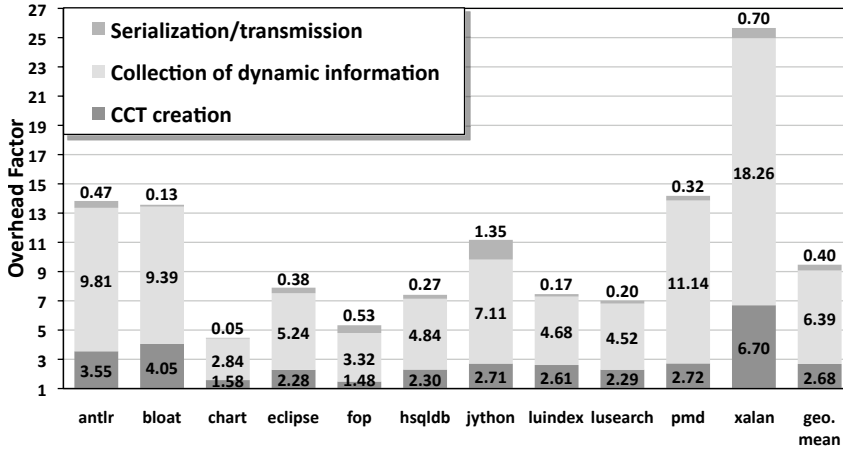

Figure 8.7: *Senseo* overhead for the DaCapo benchmarks.

benchmark completion. To this end, we modify the DaCapo benchmark harness in order to delay the end of a measurement until the transmitted data have been received and processed by the Eclipse plugin. Figure 8.7 also shows the average overhead (geometric mean) for the DaCapo suite.

On average (geometric mean), CCT creation alone causes an overhead of factor 2.68. CCT creation and collection of dynamic information result in an overhead of factor 9.07. The total overhead, including serialization/transmission, is of factor 9.47. For all benchmarks, the larger part of the overhead is due to the collection of dynamic information, where the collection of runtime type information is particularly expensive. Serialization/transmission causes only minor overhead, because in these measurement settings serialization/transmission happens only once upon benchmark completion.

Senseo features an optimized serialization mechanism that transmits the CCT in an incremental way, sending only those nodes where some dynamic information has changed since the previous transmission. Thanks to the principle of locality, typically only a small subset of the CCT nodes is transmitted. Thus, it is possible to frequently update the dynamic information in the Eclipse plugin, such as once per second.

Figure 8.8 illustrates the size of successively transmitted data packets for a single run of DaCapo's "eclipse" benchmark with a serialization/transmission rate of 1.25 packets per second.[7] Such a high serialization/transmission rate ensures that the developer always sees

[7]We chose the "eclipse" benchmark for this measurement, since it has the longest execution time in the DaCapo suite in our measurement environment.

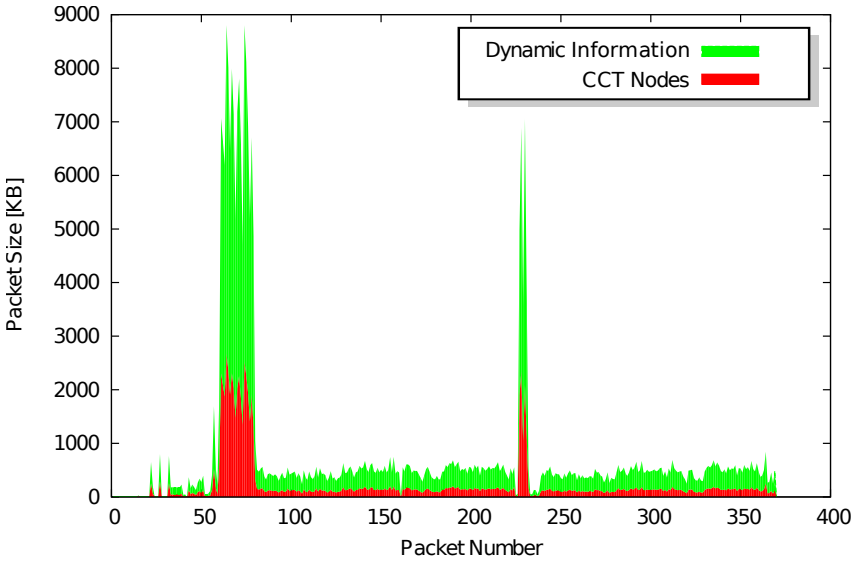

Figure 8.8: Size of transmitted data packets for "eclipse". Serialization/-transmission rate: 1.25 packets per second.

up-to-date dynamic information in the IDE, refreshed more than once per second, while the application under maintenance is running in the *MAJOR* JVM. In total, 370 packets are sent, that is, the total runtime of "eclipse" is about 296s in this setting (causing an overhead factor of 14.8, whereas a single serialization/transmission upon benchmark completion induces an overhead of factor 7.9 as shown in Figure 8.7). For each packet, Figure 8.8 differentiates between the size of the transmitted CCT nodes and the size of the sent dynamic information.

While most packets are rather small, below 1MB, some packets are considerably larger, reaching up to 9MB. The packets 60–79 appear as a major peak in the figure. We found that these packets convey dynamic information collected while the "eclipse" benchmark is compiling some projects. The minor peak in Figure 8.8 (packets 227–232) corresponds to some XML data processing. The initial packets, collecting during the startup phase of "eclipse", are very small. This can be explained by the fact that the startup phase is IO-intensive and involves much class-loading and just-in-time compilation by the JVM, which are mostly implemented in native code and are therefore not amenable to *MAJOR*'s instrumentation.

As *Senseo* can be used to gather dynamic information from all applications used in the DaCapo suite in reasonable time, we conclude that

Senseo is fast enough to cope even with large-sized applications, and it is possible to frequently transmit the collected dynamic information to the Eclipse plugin, continuously providing up-to-date dynamic information to the software developer. Even though the overall overhead is high when gathering dynamic information, we do not consider this as a major issue, as the application does not need to run at productive speed while analyzing it.

8.7 Related Work

In this section we compare related works to our *Senseo* approach.

JFluid [DMIT 04b]. Similar to *Senseo*, JFluid runs the application under instrumentation in a separate JVM, which communicates with the visualization part through a socket and also through shared memory. JFluid is a pure profiling tool, whereas *Senseo* was designed to support program understanding and maintenance. JFluid does not aggregate dynamic information but focuses on presenting behavioral information about a specific system execution.

Ferret [DE A 08] integrates a query tool into Eclipse to allow developers executing conceptual queries about source artifacts directly in the IDE. An example of such a query is "callers of method x". Ferret also takes into account dynamic information In contrast to *Senseo*, Ferret does not aim at giving an overview of the system or enriching the static IDE perspectives with dynamic information; it present the results of queries in a view separated from the source code.

Compass [LIEN 09] is a back-in-time debugger for the Smalltalk IDE. As already mentioned in Section 7.6, Compass does not integrate the reified dynamic information in the conventional source code views and focuses on a specific system execution. Moreover, the reified information is volatile and is discarded at the end of a debugging session.

Fluid source code views [DESM 06] are similar to *Senseo* as they also link call sites in code to invoked methods in an IDE's source views. However, these links are statically determined, so they might be imprecise or even incorrect for polymorphic call sites. The comparison between *Hermion* and fluid source code views in Section 7.6 concerning enhancing the source views basically also holds for *Senseo*, with the difference

that *Senseo* extends the tooltip in Eclipse's source editor, while *Hermion* and fluid source code views insert icons and, in the latter case, even the invoked source code directly in the source editor.

Approaches analyzing development activities. As mentioned in Section 2.1.3, there are several proposals analyzing development activities recorded in the IDE to support navigation and maintenance of software systems. FEAT [ROBI 03a] identifies concerns from recorded program investigation activities performed in the IDE and visualizes these concerns with graphs. NavTracks [SING 05] recommends source entities related to the currently selected entity by analyzing how developers navigated and modified the system in the past. Mylyn [KERS 05, KERS 06] exploits programmer activity to build a degree-of-interest model for the program elements in a system and highlights the elements considered interesting for the task-at-hand. *Senseo* is similar to these works as it also relates source artifacts to each other, but based on how these artifacts correspond to each other at runtime. *Senseo* focuses on the dynamic relations between source artifacts and not on how these artifacts have been navigated or modified by developers. Moreover, *Senseo* also supports developers in gaining an overview of the behavior of a system, which is not supported by the FEAT, NavTracks or Mylyn. *Senseo* differs in the same way to proposals that exploit the source history to find dependencies between artifacts as these approaches such as ROSE [ZIMM 04a] or Hipikat [CUBR 03] reveal how artifacts have been committed together to the source repository but not how they communicate at runtime.

Visualizations of dynamic information not integrated in IDEs. In Section 2.2.2 we discussed several techniques visualizing dynamic information outside the IDE in a separate tool. Examples for such tools and environments are Jinsight [DE P 93], Shimba [SYST 01], Collaboration Browser [RICH 02], GraphTrace [KLEY 88], or Jive [REIS 03] and Jove [REIS 05].

Senseo differs from these related works as it integrates and embeds dynamic information directly in the IDE locally to specific static system artifacts instead of providing a general overview in a separated tool. Such a local integration particularly recognizes the conceptual relation between static and dynamic aspects of software systems. Hence, dynamic information readily made available by *Senseo* in the IDE allows developers to embrace such information while navigating the software space and while working with source code. It is this tight integration of dynamic information in the conventional source perspectives of an IDE, without further increasing the information overload, which supports develop-

ers in software maintenance, as the results of the conducted controlled experiment confirm (cf. Section 8.5).

8.8 Summary of the Chapter

In this chapter we presented *Senseo*, an approach for gathering and integrating various kinds of dynamic information from running Java applications within the Eclipse IDE. The provided dynamic information includes callers, callees, runtime type information, method invocation counters, and object allocation metrics. *Senseo* integrates dynamic information in the package tree, the ruler columns, and in the source editor tooltips of the Eclipse IDE. In addition, *Senseo* offers a condensed and interactive visualization of the CCT and provides a navigable view on all dynamic collaborators of a source artifact (package, class, or method). The dynamic information is continuously updated in the IDE while an application is running.

A controlled experiment with 30 professional developers confirms that the dynamic information provided by *Senseo* significantly improves correctness and reduces the time needed for various software maintenance tasks. A performance evaluation shows that our approach is practical and able to visualize dynamic information in the IDE that is updated more than once per second.

Senseo addresses the shortcomings of *Hermion*, that is, it represents dynamic collaboration at a higher level (package and class level) than just presenting runtime types or receiver types of message sends on a source code level. Moreover, *Hermion* does not provide an overview of dynamic collaboration as *Senseo* does with its collaboration view. However, even *Senseo* does not visualize dynamic collaboration patterns between packages, classes or methods, instead it presents such information in lists. This often makes it hard to quickly grasp the "big picture view", that is, an overview of how and how often artifacts communicate with each other at runtime, for instance how many classes of two packages communicate with each other and which methods are thereby invoked. This problem is tackled by *CollView* which we present in the following chapter. Another shortcoming of *Hermion* which is neither addressed by *Senseo* is the missing representation for software features in the IDE. This problem is tackled by *FeatureEnv*, which is the topic of Chapter 10.

Chapter 9

CollView – Representing Dynamic Collaboration in IDEs

9.1 Introduction

9.1.1 Positioning *CollView*

In this chapter we introduce *CollView*, an enhancement to the Smalltalk IDE integrating visualizations of dynamic collaboration. *CollView* aims at fixing the shortcomings of *Hermion* and *Senseo*. These two approaches do not visualize collaboration patterns in easy to understand visualizations. They just link collaborating artifacts in the source code views instead of providing an overview of dynamic collaboration between artifacts to the developer.

CollView represents dynamic collaboration patterns between conceptually related but statically distributed artifacts by visualizing such collaboration in charts embedded in the IDE. Moreover, *CollView* supports developers in navigating dynamic dependencies between source elements from a high package level down to a class or method level to gain an understanding of the dynamic behavior of a system. The collaborations charts generated by *CollView* also make visible the execution flow in the system, for instance which methods invoke with other methods in which order. Additionally, *CollView* tackles the problem of hidden execution

paths between source elements, at least on a method level, and to some degree also on a class or package level. Eventually, *CollView* gives an overview of a system by visualizing on a high package level the communication occurring between all system packages. Figure 9.1 briefly summarizes the IDE problems *CollView* primarily addresses.

	Activity / Problem	Feature investigation	Feature implement-ation	Artifact investigation	Dependency investigation	Runtime interaction investigation	Artifact usage investigation	Execution pattern investigation	Quality assessment	Domain concept analysis
Over-loaded views	No overview	✔	✔		✔				✔	
Static views	Distributed artifacts	✔		✔	✔	✔		✔	✔	✔
	Collaboration hidden	✔		✔	✔	✔		✔	✔	✔
	Execution paths hidden			✔	✔	✔	✔	✔		

Figure 9.1: *CollView* aims at explicitly representing and visualizing dynamic collaboration between related but statically distributed source artifacts. Moreover, *CollView* uncovers execution flow primarily on a method level but to some degree also on a class or package level. Eventually, *CollView* contributes to a better system overview by displaying collaboration on a package level.

The rest of this chapter introduces *CollView* and motivates the need to have a representation of dynamic collaboration in IDEs. Afterwards, we report on how *CollView* visualizes collaboration between source artifacts in charts and how these charts are integrated in the IDE. We validate *CollView* by means of performance benchmarks for the data gathering technique and by reporting on user feedback.

9.1.2 Introduction to *CollView*

When maintaining software, developers typically navigate a system's source code in a development environment (IDE) to gain an understanding of how the system functions. Source code browsing alone does not answer many of the questions about how an object-oriented system behaves at runtime. Object-oriented system behavior stems from the dynamic cooperation of interacting classes and methods [WILD 93]. IDEs traditionally provide views to support reasoning about static source artifacts. There is a lack of solid support for behavioral information within the IDE forcing developers to themselves build up and maintain a mental map of the dynamic relationships between source artifacts.

A common practice of object-oriented software development is to incorporate documented design patterns [GAMM 95] to solve well-known, recurring design problems. Design patterns are a kind of "micro architec-

tures" consisting of static program artifacts and dynamic collaborations between them. The *Chain of Responsibility* pattern, for example, processes a series of objects, involving several different static source artifacts whose dynamic interaction is often difficult to uncover from the source code. While patterns may increase the flexibility of the system, they usually also introduce a level of complexity, making a system even more difficult to understand just by static source code browsing.

IDEs such as Eclipse [ECLI 03] provide plugins to generate visual representations of a system's behavior (*e.g.* UML sequence diagrams). These representations are usually restricted to providing pure snapshot visualizations and lack interactive capabilities to support navigation of the software artifacts directly within the IDE. Moreover, they often rely on trace-based post-mortem analysis. Developers accustomed to immediate availability of information in the IDE are less likely to incorporate such analyzes in their daily work.

Developers typically focus on specific static artifacts, *e.g.* key classes which they have identified as relevant for their current task, and have a need to study their dynamic relationships while source code browsing. Immediate access to visualizations of dynamic class relationships that evolve in synch with the static artifacts would provide the developer with the missing behavioral information. To offer real added value to a developer, visualizations should provide a means to navigate through and browse the source code of collaborating artifacts. Additionally, such visualizations have to be lightweight and easy to learn and understand in order to not overload developers with even more complex information.

In this chapter we propose the introduction of dynamic collaboration representations that are readily accessible in the IDE as interactive, navigable views. To achieve this goal, we face several challenges. In particular, we focus on three key research questions:

- *Why is it crucial to understand how source artifacts dynamically communicate from within the IDE?*
- *How can we achieve the immediate availability of dynamic information in the IDE?*
- *How do we browse and reason about dynamic collaborations in an IDE without further overloading the environment with information?*

We address these questions in detail in Section 9.2. In Section 9.3 we contribute our working prototype called *CollView* to illustrate how to represent, visualize and navigate dynamic collaborations directly in the IDE. *CollView* is available for the Squeak and Pharo Smalltalk IDE. In Section 9.4 we validate the usefulness of this approach by studying two

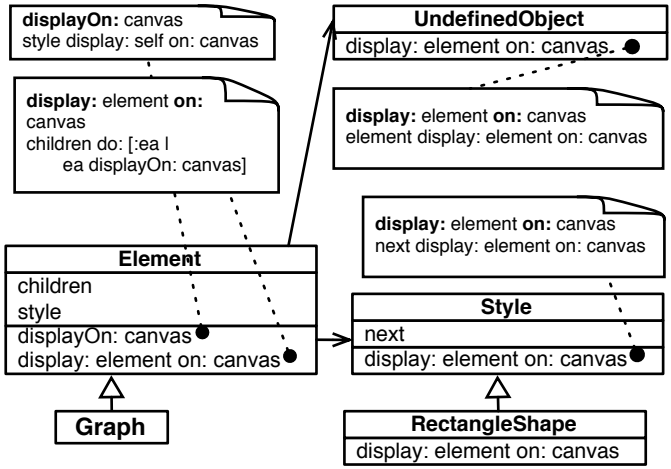

Figure 9.2: UML diagram of Mondrian classes involved when displaying a graph.

applications where the behavior is not easily understood by source code reading. Furthermore, we ask developers to assess the gained benefit of using embedded dynamic analyzes and our dynamic collaboration representations in *CollView* when trying to understand these software systems. We present a critical analysis of our proposal in Section 9.5, provide an overview of related work in the context of our work in Section 9.6 and conclude in Section 9.7.

9.2 Hidden Dynamic Collaboration

The dynamics of object-oriented software systems are hidden in development environments. Widely used language concepts such as polymorphism or dynamic method binding make it difficult to understand the behavior of object-oriented systems. To fully comprehend the workings of behavioral design patterns [GAMM 93] such as *Chain of Responsibility*, *Visitor*, *Observer*, or *State*, the developer needs to understand how the delegation between the participating objects works at runtime. Static analyses can reveal inheritance relationships or method call definitions, but to which objects messages are sent and which methods are actually invoked is not visible prior to execution. This implies that only through dynamic analyses we are able to reveal this information.

We illustrate the problem of missing dynamic information by taking as an example the task of trying to understand a concrete incarnation of the *Chain of Responsibility* pattern in Mondrian [MEYE 06], a graph rendering software and one of the case study systems of our work. We chose the *Chain of Responsibility* pattern because it is a frequently used pattern in practice [VOK 04] and well explains the problems we are addressing in this chapter. To define a graph of nodes and edges in Mondrian, a developer implements a script specifying the layout and styles of nodes and how they are connected to each other with edges. The following code illustrates a simple script to render a tree graph for a class hierarchy visualization:

```
graph nodes: aClass withAllSubclasses.
graph edges: aClass inheritance using:
   (Line from: #superclass to: #subclass).
graph layout: TreeLayout new.
graph style:
  (RectangleShape new color: #linesOfCode),
  (Extent new width: #numberOfAttributes;
              height: #numberOfMethods).
```

Mondrian has a generic internal design centered around the class *Element*. The entire graph itself is a subclass of *Element* as are nodes and edges. Different styles to be applied to nodes are composed and arranged in a *Chain of Responsibility*. This means that after being applied, the first style passes to the next style in the chain until all of the defined styles have been applied. As an element can be a graph, a node or an edge, it is virtually impossible to tell from reading the source code what methods are actually invoked at runtime. Elements can also have children, *e.g.* in a class hierarchy graph the subclasses of a given class are modeled as child elements of the element representing that class. The static analysis of Mondrian results in the UML diagram displayed in Figure 9.2. Modern IDEs (*e.g.* Eclipse [ECLI 03] or Squeak [INGA 97]) can often automatically generate UML diagrams from source code, for instance with the inCode plugin [INCO 09].

However, the static UML diagram does not shed much light on how the entire graph of all the nodes and edges is displayed at runtime. The actual dynamics of Mondrian displaying a graph is illustrated in Figure 9.3: To render the whole graph the method *Element » displayOn: canvas* is invoked on the graph object. This method triggers the traversal of the chain of styles. The last style in this chain triggers the displaying of the element itself, *i.e.* invoking method *Element » display: element on: canvas* which iterates over all children of that element and invokes method *Element » displayOn: canvas* on them. The style of the graph itself is undefined. An instance of *UndefinedObject* hence directly invokes *display: element on:*

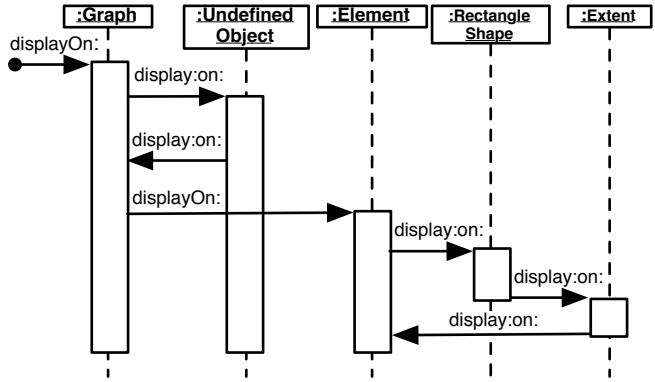

Figure 9.3: Sequence diagram in Mondrian to display a graph.

canvas to iterate over the graph's children. Revealing the interplay of graph, nodes and styles together at runtime is not feasible by studying a static UML diagram or the static source code, in particular as both graph and elements are simply called *element* in the source code. Moreover, it is not obvious that relevant behavior implemented in *UndefinedObject* is intended to be invoked if an element's style is undefined.

Our working example from our Mondrian case study is a typical incarnation of the type of challenges a developer faces when trying to reverse-engineer a software system within the IDE: by performing manual or semi-automated static analysis, she discovers relevant static source entities such as packages or classes (*e.g.* the classes *Element* and *Graph*) but fails to see from the source code or from available UML diagrams how the static artifacts interact with each other at runtime, *e.g.* which dynamic dependencies exist between them. The dynamics of source entities, *e.g.* collaboration between objects, are not explicitly available in IDEs. Although techniques to reverse engineer sequence diagrams using dynamic analysis (*e.g.* Briand *et al.* [BRIA 06]) and also IDE plugins generating these sequence diagrams (*e.g.* MaintainJ[1] for Eclipse) exist, they do not offer a means to browse and navigate the source artifacts depicted in these diagrams. Thus these views are not tightly embedded in IDEs. They also fail to provide the information immediately after system's execution. Moreover, these sequence diagrams generally cannot deal with large amounts of runtime data.

In the following we elaborate on concepts to increase the understanding of dynamic collaborations between static entities directly from within the IDE by making them explicit. The goal is to give developers easily

[1] http://www.maintainj.com/

comprehensible views used to focus on specific executions and specific artifacts.

9.3 Representing Dynamic Collaboration in the IDE

To represent dynamic collaboration between various source artifacts explicitly within the IDE, we first need to execute the system under study to gather runtime information. In a second step we empower developers to reason about the runtime information within the IDE by representing the dynamic information as interactive and navigable views. The chief goal is that developers select arbitrary static artifacts within the IDE, *e.g.* several classes or all packages of an application, whose dynamic collaboration they need to comprehend for a given usage scenario. Developers then execute the system (exercising specific features of interest), and the IDE takes care of the dynamic data gathering and immediately presents this data in the form of views to developers.

9.3.1 Gathering Dynamic Information

As we realized in previous chapters, for instance in Chapter 2 or Chapter 7, analyzing the runtime behavior of applications using tracing tools is time-consuming and generates large amount of data. This makes such tools inappropriate for integration in IDEs as developers require immediate benefit from the results of dynamic analyzes. Partial behavioral reflection overcomes these problems as it enables us to select in a very fine-grained manner on what dynamic parts of a system we want to reflect. Thus, we rely on the same approach to dynamic data gathering as we use for *Hermion* (cf. Section 7.3); this approach is called Reflectivity. For *CollView* we can reuse the same interface to Reflectivity as used in *Hermion* as both tools need to extract the same information from a running application. However, *CollView* does not exploit runtime type information, thus we do not reify such information but focus on the reification of message sending.

To reveal how a Mondrian graph is rendered, the developer needs to understand how the classes *Graph*, *Element* and *Style* depicted in Figure 9.2, communicate at runtime, *i.e.* how they send messages between each other. Using the Reflectivity framework we specify that only dynamic information concerning message sends occurring in these three classes should be collected, while all other dynamic data, *e.g.* message sends to system classes or variable accesses, are ignored.

The developer triggers the dynamic analysis of the entities of interest directly from within the IDE and then runs the system, either using recorded scripts such as test cases or by directly running the system as an end-user. Reflective behavior, which is introduced into the binary of methods, collects information about every message send occurring within the selected entities. For more details of how we build our IDE enhancements on partial behavioral reflection, we refer the reader to *Hermion* introduced in Chapter 7.

9.3.2 Explicit Dynamic Collaboration

CollView enables the developer to browse dynamic collaborations between static source artifacts as soon as any dynamic information has been gathered. *CollView* provides *interactive collaboration charts* that support browsing and analysis of dynamic collaboration. We embed these charts tightly in the IDE so they are directly accessible to a developer working on the source code: In the case of our Mondrian example, a developer selects the static entities of interest, *e.g.* class *Graph*, *Element* and *Style* to observe their dynamic behavior, exercises specific features of Mondrian, and even while the system is running, she can open an *interactive collaboration chart* showing the dynamic communication occurring between instances of the selected classes.

Interactive Collaboration Charts

In this section we describe the details of our *Interactive Collaboration Charts*. All our views are graph representations of dynamic collaborations at different levels of granularity. Their interactive capabilities support navigation of source code artifacts. Furthermore, we map information (*e.g.* number of message sends) to edges and nodes, similar to the polymetric views for visualizing runtime information described in the work of Ducasse *et al.* [DUCA 04].

Class Collaboration Chart. This chart is conceptually similar to UML's sequence diagram (*e.g.* Figure 9.3). We display in a class collaboration chart how messages are passed between objects of classes. As sequence diagrams do not scale for larger applications with a deep nesting level of message sending involving many objects, we condense the information in the collaboration chart to show each selected class and each message sent between instances of these selected classes only once. We take into account indirect communication, *i.e.* if an instance of class A sends a message to an instance of a class not selected, but an instance of this class

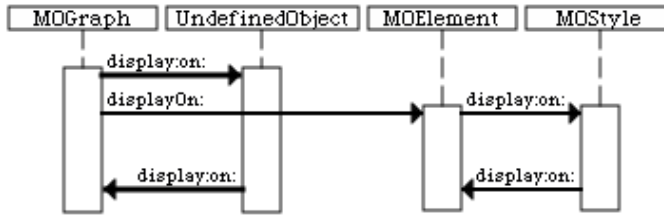

Figure 9.4: Class Collaboration Chart generated by the IDE.

sends a message to objects of any selected class, we show a dashed line between the two selected classes denoting indirect communication. The order of message sends is not preserved in this view (the same message sent between any instances of two classes is displayed as one single edge, no matter how often it occurs). As a result, the collaboration chart does not become too cluttered even for large systems. To further compress the dynamic information we adopt an approach similar to that described in the work of Hamou-Lhadj and Lethbridge [HAMO 03]: we ignore repeated message sends occurring as a result of loops or recursions when calculating the message send frequency. The views guide developers to understand how selected artifacts interplay at runtime without confronting them with too much information.

CollView is also capable of rendering a full-fledged UML sequence diagram on demand through interaction with the *Class Collaboration Chart* (for instance, by selecting two classes), if the developer wants to study a particular collaboration in more detail.

Once again considering our Mondrian example, performed by the three classes *Graph*, *Element* and *Style*, we show the interactive collaboration chart in Figure 9.4. All messages exchanged by these classes, including *UndefinedObject*, are visually presented. The developer can convert the whole chart into an UML sequence diagram or select a specific message send and open a new class collaboration chart with this message send as a starting point. Additionally, it is also possible to open an interactive collaboration chart on a specific method, *i.e.* to see all collaborations between this method and other methods. Clicking on the edge *#displayOn:* leaving *Graph* for instance brings up the method collaboration chart shown in Figure 9.6.

Package Collaboration Chart. We provide a big picture view of dynamic collaboration at the package level, typically representing the entire system, if necessary even including system packages. We consider that a dynamic collaboration exists if an instance of a class in one package

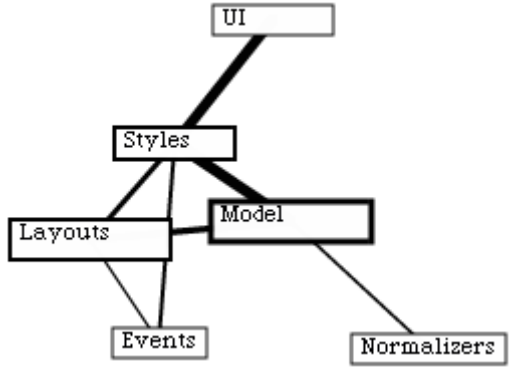

Figure 9.5: Package Collaboration Chart generated by the IDE.

sends messages to an instance of a class in the other package. In this case our chart displays an edge between the packages. An example of a *Package Collaboration Chart* is shown in Figure 9.5. The developer can study the collaboration between any two packages by clicking on the edge to discover which classes actually communicate with each other. It is then possible to open a *Class Collaboration Chart* on any two collaborating classes to study the collaboration on a message sending level.

Method Collaboration Chart. At a fine-grained level of detail we want to reason about the method level, *i.e.* with which methods a given method communicates at runtime. For this purpose we use a *method collaboration chart* focusing on a particular method. This chart shows all messages sent from within this method as edges that invoke other methods. Once again we map frequency of invocation to the thickness of the edges and the number of invocations relative to the overall number of invocations to the size of the nodes. Figure 9.6 presents the method collaboration chart for the method *Element » displayOn: canvas* of our Mondrian example.

A key characteristic of our collaboration charts is that they are interactive and support browsing within the IDE. Clicking on any class in a class collaboration chart opens a source code view of this class. By clicking on a message send between two classes (*i.e.* an edge) the user can, for example, see all methods invoked by this send or open method collaboration charts with the invoked method as a root. A click on the message *displayOn:* sent from *Graph* to *Element* as shown in Figure 9.4 opens a method collaboration chart focusing on the method *Element » displayOn: canvas* shown in Figure 9.6, as *Graph* simply uses the method *displayOn:* implemented in its parent class *Element*.

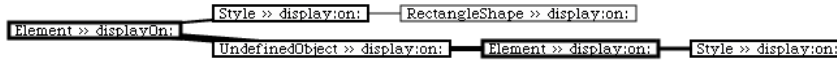

Figure 9.6: Method Collaboration Chart generated by the IDE.

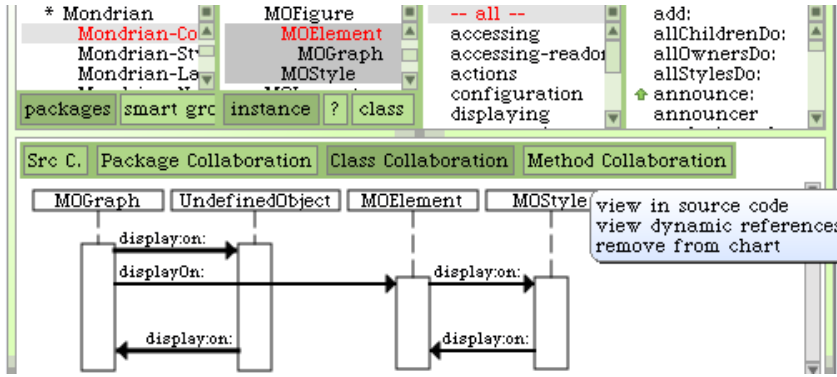

Figure 9.7: Integration of a class collaboration chart in the Squeak Smalltalk IDE.

Additional dynamic information is available on demand in collaboration charts, *e.g.* the average execution time of a message send, the number of times a message has been sent, the number of instances of a class, or the number of classes from two different packages that communicate with each other, and so on. Such information may prove useful to assess or identify performance bottlenecks. We also provide visual means to quickly identify frequent communication paths by displaying the edges in this path more thickly, as shown in Figure 9.5 or Figure 9.6. Finally, the charts are dynamically modifiable, *i.e.* the developer can for instance remove static artifacts from the chart or add additional artifacts (*e.g.* classes) that should also be taken into account when rendering the chart.

In the following, we explain how these charts are tightly embedded in the IDE, *i.e.* how these charts integrate with the static views of source artifacts such as packages, classes or methods.

9.3.3 Enhancing Existing IDE Tools

Typically, an IDE provides means to browse source code entities in a top-down manner, *i.e.* going down from packages, classes to single methods, *e.g.* by using a tree view. In the Squeak Smalltalk IDE [INGA 97] packages,

classes, method categories and methods are navigated in columns in this order, as visible in Figure 9.7. We enhance this source code view in the Squeak Smalltalk IDE with means to select several static artifacts of the same kind (*e.g.* several classes). The IDE instruments these selected entities on demand with partial behavioral reflection as described in Section 9.3.1 to gather dynamic information, without having to halt the system if it is already running. The developer then executes a feature of interest in the subject system or runs a particular test case to generate runtime data to be studied in more detail. Within the IDE the developer chooses the class collaboration chart to view this chart based on the previously gathered information.

Such a scenario is illustrated in Figure 9.7 where the classes *Graph*, *Element* and *Style* have been selected. After the program, *i.e.* Mondrian, has been executed, selecting the tab "Class Collaboration" brings up the class collaboration chart for these three classes as shown in Figure 9.4.

In all charts, static artifacts are navigable, *e.g.* selecting a class opens this class in parallel to the chart (see actions of class *MOStyle* in Figure 9.7), or clicking on edges in class charts brings up the method invoked. Modifying source code marks all charts involving this source code as obsolete, *i.e.* these charts need to be updated by a new run of the system.

This approach of integrating collaboration visualizations in IDEs is very lightweight and does not further overload the busy interfaces of development environments as these charts only appear on demand, when the developer wants to investigate in detail the dynamic collaboration between particular source artifacts.

9.4 Validation

First, we validate *CollView* by reporting on the efficiency and performance of the dynamic information gathering and the generation of the charts. Second, we present the results of a study conducted with developers, where we asked them to assess the added value of having access to our dynamic collaboration views tightly integrated in the IDE.

9.4.1 Performance Benchmarks

We ran two benchmarks evaluating the generation of *Class Collaboration Charts*: First, we selected the three classes of our Mondrian example, (*Element*, *Style* and *Graph*), to generate a chart highlighting how these classes interact at runtime. To gather the runtime information, we let

Mondrian generate ten complex graphs each with a hundred nodes. We measured the time to execute our scenario once with and once without dynamic information collection activated. We also measured the time to render the charts after having collected the dynamic information. We summarize the results in Table 9.1.

As a second case study we analyzed Pier [RENG 06], a sophisticated, web-based content management system implemented in Smalltalk. Pier has a generic model to provide its services, thus we decided to analyze its core package called *Pier-Model* in which the main classes reside. In total, we analyzed eight classes in this package which we have chosen randomly, without taking into consideration any knowledge about how these classes communicate with each other. The selected classes consist of 215 methods in total. After analyzing the behavioral information gathered while exercising a Pier feature *edit page*, we noticed that these eight classes indeed directly or indirectly collaborate at runtime, resulting in a huge collaboration chart. Again, we measured the time to run this feature with and without runtime analysis, and the time to render the chart. We present the results in Table 9.2.

The figures we obtained from our benchmarks lead us to conclude that the information gathering technique is efficient enough for most practical use cases where only a limited number of classes are to be analyzed. As the comparison between the Mondrian and the Pier benchmark illustrates, the analysis time significantly increases in proportion to the number of analyzed source artifacts. We also performed a preliminary benchmark analyzing all classes of Pier, which resulted in an overhead of more than factor seven. It is possible to perform such analyses covering all application classes, but the resulting class collaboration chart will be too cluttered to be of use to developers. This kind of chart is typically intended to only cover a subset of all the application classes. In such a case we do not consider the performance of data gathering to be an issue.

We note that gathering information to generate package collaboration charts for a whole application is at least as efficient as generating class charts for a limited number of classes as much less information has to be gathered to reason about package dependencies, basically only whether any two packages exchange methods and how often such methods are invoked. Analyzing package dependencies of the whole Pier application results in an overhead of less than factor two.

Drawing the charts is very efficient even for large charts, as illustrated by the results of both benchmarks.

Action	Measured time (ms)
Execution w/o data collecting	7246
Execution w/ data collecting	20725 (overhead 186%)
Chart rendering	0.2

Table 9.1: Time to gather data and render a class collaboration chart for Mondrian.

Action	Measured time (ms)
Execution w/o data collecting	56
Execution w/ data collecting	233 (overhead 316%)
Chart rendering	0.3

Table 9.2: Time to gather data and render a class collaboration chart for Pier.

9.4.2 Developer Feedback

We asked several Smalltalk developers familiar with the Squeak Smalltalk IDE to use our charts while working on a feature enhancement request in a software system that was unknown to them before. All subjects of this preliminary user experiment were graduate students with at least one year of Smalltalk experience and several years of programming experience in other object-oriented languages such as Java. To realize the enhancement of the subject system (namely the Pier system), the study subjects needed to gain an understanding of at least five classes in this system, in particular how these classes collaborate with each other. We did not mention which classes are important to realize the enhancement, but explained the model package of Pier in which the feature enhancement had to be implemented by means of a static UML class diagram not revealing any information about the runtime collaboration between the various classes.

The task for the developers was first to gain a basic understanding of the dynamics between the classes in the model package of Pier and then second to focus on the enhancement request, which was relatively easy to perform as soon as the classes realizing the feature have been identified. Concretely, we asked them to implement a move mechanism for an existing web page in the content management system. A copy and a remove mechanism were already present, which we mentioned in the explanation phase. We also gave the hint to implement the move feature by combining the copy and the remove features. The subjects had to find out how the commands realizing these actions and the classes representing web pages interact at runtime to be able to implement the feature request. We advised them to run the copy and remove features

Statement	Average rating
Strong effect on class collaboration understanding	4.4
Strong effect on general understanding of the feature	4.1
Strong effect on complete system understanding	3.2
Strong effect on execution overview	4.3
More efficient navigation of source entities	3.4
Faster identification of relevant source entities	3.9

Table 9.3: User rating for asked statements during the experiment.

of Pier to generate a class collaboration chart expressing the dynamic relationships between the model classes contributing to these features. This scenario makes use of three design patterns that subject had to grasp, the Command, Composite, and Observer patterns.

After the experiment, we asked several questions to the subjects to be answered on a Likert scale [LIKE 32] from '1' to '5' where '1' means 'strongly disagree' and '5' means 'strongly agree' with the asked statement. The statements we issued to the subjects and the average rating we obtained are shown in Table 9.3. Additionally, the subjects gave some comments, *e.g.* that the charts were very useful to locate relevant static entities to be studied further and to understand how they relate to other entities at runtime, leading to a faster comprehension of the executed features.

This developer feedback constitutes a preliminary empirical evaluation of our work. Nonetheless, the feedback we obtained encourages us to conduct a full-fledged empirical evaluation covering several tasks to be performed by the subjects in order to obtain quantitative data validating the effect of the interactive collaboration charts on program comprehension more thoroughly. We expect that *CollView* also supports well typical software maintenance tasks such as debugging, fault isolation and correction, or feature identification, as all these different tasks encompass hidden dynamic collaborations between source artifacts made visible by *CollView*.

9.5 Discussion

In this section we discuss several aspects of our work which we consider to be critical to our approach. These aspects cover four dimensions of our proposal: (1) efficiency and (2) completeness of dynamic analysis, (3) usability, and (4) extensibility of the interactive presentation of the

information, *i.e.* collaboration charts.

Efficiency. We have provided the results of our benchmarks on pure data gathering efficiency in Section 9.4.1. However, we need to discuss scalability of our approach in terms of size of the subject system or the amount of the resulting dynamic data. Concerning the size of systems we can analyze with our approach, we need to distinguish between class collaboration and package collaboration charts. As shown in the previous section, viewing a full-fledged class collaboration chart covering the whole system does not provide much value to the developer as it is not easy to locate any concepts in the resulting view. The package collaboration chart is better suited when reasoning about the dynamic collaborations of the entire application as it provides useful information about frequent communication paths between the different packages. From this view the developer can identify communication paths of interest, which she can then study further on the level of class collaboration. By starting to analyze package collaboration instead of class collaboration the developer can also reason about larger systems in which several thousands of message sends occur to realize a specific feature. We validated this by studying such an execution of a system (*i.e.* the test suite of Pier consisting of more than 1000 test cases).

Another way to deal with a large amount of data is to adopt filtering and compression techniques. As mentioned in Section 9.3.2 we filter out recurrent method calls resulting from loops using a similar algorithm as presented by Hamou-Lhadj *et al.* [HAMO 03] before finally storing the information used to generate class collaboration charts. As developers typically select specific source entities, *e.g.* classes, to be analyzed dynamically, all information not generated by the selected entities is omitted by default, which is useful to ignore much data not required for a given use case from the start.

Completeness. Dynamic analysis always raises the issue of coverage and completeness of the obtained dynamic information [BALL 99]. The results of dynamic analyses depend on what exactly gets executed in the system under study. The collaboration between several classes for instance can vary heavily between two different features being executed. We recognize this fact, but in the context of an IDE, developers normally focus on very specific tasks, *i.e.* correcting a defect or enhancing a feature. For this kind of task, it is an advantage rather than a drawback to focus on specific executions of the system instead of having complete coverage of all theoretically possible executions. Being able to tailor the view to a

very limited scope, *e.g.* executing a feature with pre-defined input values to reproduce a reported bug, eases the understanding for this specific scenario as noise of other executions are not included in the collaboration charts.

Collaboration charts contain less information than for instance sequence diagrams, as the order in which messages are sent is not preserved. Collaboration charts provide an overview to quickly grasp information of interest without distracting developers by displaying too much information. As soon as they have identified communication patterns of interest, they can study them more in detail, by using a full-fledged sequence diagram, a debugging session or simply reading the source code.

Another completeness issue arises due to the fact that the static source artifacts are selected by the developer. This is inherently error prone as she might omit some of the entities relevant for the problem at hand, *e.g.* a defect in a system's feature. This is in particular true when dealing with a large, unknown system where the important entities have to be identified first. However, in many cases the developer has already a basic understanding of the system in terms of important entities to be considered in the first step. As the package collaboration chart allows the developer to start the analysis at a coarse-grained level, she can start with this initial knowledge and gradually select other entities of interest. In Mondrian we discovered the important role of the *UndefinedObject* class by first studying the package collaboration where we could easily see that Mondrian communicates with the system package containing *UndefinedObject* (*i.e.* package *Kernel-Objects*).

Usability. Usability is a fundamental aspect of a development environment. We constantly integrate user feedback in our own development process of *CollView*. In particular, we collected user feedback in early development stages to assure that developers understand and can effectively use our interactive collaboration charts within the IDE. From the discussions with developers, *i.e.* end-users of our work, we constantly obtained hints and remarks about missing features and better integration of our work into the existing environment of the IDE, *i.e.* Squeak Smalltalk. We consider these feedback loops to be a best practice when working on enhancements to development environments, in particular to increase usability, but also to discover potentially useful features. This is even more important when working on tools incorporating dynamic information, as the vast amount of data makes it difficult to devise representations that are easy to grasp.

Extensibility. Currently we have built *CollView* in the Squeak Smalltalk environment [INGA 97], as this IDE is easily extensible. The concepts we describe in this chapter, however are generic and we do not expect it to be difficult to port this work to other IDEs such as Eclipse [ECLI 03] for Java. Making the various charts interactive is straightforward. More challenging is the efficient collection of data without having to recompile or otherwise prepare the subject system. Reflex [TANT 03] provides similar means for partial behavioral reflection as adopted in our approach, but it is less efficient and requires the reflective behavior to be anticipated before starting the system. As the IDE takes care of the correct installation of the reflective data gathering in the subject application before start up, this is no drawback in many cases.

The integration of the charts into the IDE, *i.e.* how to interact with the charts from and back to traditional IDE tools, is easily extensible and adaptable, provided that the IDE offers open frameworks to extend itself, which is the case for both the Smalltalk and Eclipse IDE.

9.6 Related Work

In this section we compare *CollView* to existing work in the context of IDEs, representing dynamic collaborations and visualizing behavior of object-oriented systems. Most of these related works have been introduced in Section 2.2.2.

Collaboration Browser [RICH 02] represents a program's behavior in terms of collaboration patterns. This approach performs postmortem analysis on execution traces of a system's behavior and represents the collaborations in visualizations similar to UML sequence diagrams. These visualizations are integrated in a minimal browser separated from the conventional IDE; the visualizations can be queried by the developer. No interaction with the static view on the system is possible, *i.e.* developers cannot use this browser to maintain the system.

Cornelissen *et al.* [CORN 07b] propose to use sequence diagrams to visually display the behavior of test suites. They address scalability issues and propose abstractions to efficiently represent the trace data. They suggest limiting stack depth and omitting constructors to filter out interactions that are not needed for comprehension. We also address the issue of limiting the amount of information to be represented in the views by means of partial behavioral reflection and filtering techniques as discussed in the previous section.

Polymetric views for condensed runtime information [DUCA 04] give an overview of the communication between classes in the whole system. These views are provided by a tool separated from the IDE, thus no interaction between these views and the static source code is possible. Moreover, focusing on specific artifacts and looking at how they communicate with the rest of the systems is not supported by this approach.

Program Explorer [LANG 95] visualizes the dynamic communication of programs and focuses on identifying and visualizing design patterns. Unlike *CollView*, Program Explorer does not show complete information concerning the communication between source artifacts but focuses on information considered to be relevant to recover design patterns in programs. Moreover, Program Explorer is not able to show how a specific artifact collaborates with the rest of the system. Program Explorer is a stand-alone tool not integrated in any IDE.

Jinsight [DE P 93] focuses on the visualization of runtime interactions between objects with the goal of easing the identification of performance bottlenecks, for instance two objects heavily communicating with each other. Contrary to *CollView*, Jinsight does not provide an overview of the collaboration patterns in a system and is not able to visualize how specific artifacts collaborate with other artifacts such as packages, classes, or methods. Jinsight is a tool separated from the IDE.

GraphTrace [KLEY 88] presents animated graphs of runtime activity and thus highlights the current activity of a program. The runtime graphs are usually very large, which makes it hard to integrate such graphs in an IDE and to use them during software maintenance. GraphTrace is not able to provide an overview of the communication occurring between system artifacts.

Shimba [SYST 01] is an environment for reverse engineering Java systems by providing scenario diagrams representing execution traces. These traces are usually very large and thus Shimba, as GraphTrace, is hard to embed or use in an IDE in order to support program comprehension and software maintenance.

Jive and Jove [REIS 03, REIS 05] visualize the runtime activity of Java programs in real time with the goal of supporting software development activities such as debugging and performance optimizations but also understanding of runtime behavior in general. The focus of *CollView* is to

integrate visualizations of runtime collaborations in the IDE to provide useful insights and interactive navigational aids for developers while working with the source code of a system. This means that *CollView* not only visualizes the runtime behavior, but uses the visualizations to navigate the system, hence its aim is not solely to boost understanding for the software, but to use the visualization of runtime collaboration to evolve or maintain a system in the IDE.

9.7 Summary of the Chapter

In this chapter we proposed to explicitly represent dynamic collaborations between static artifacts with *CollView*. We addressed the following questions throughout the chapter:

- *Why is it crucial to understand how source artifacts dynamically communicate from within the IDE?* We elaborated on use cases where an explicit representation of dynamic collaboration in IDEs is useful for developers, for instance to understand the interplay of several classes in a system or to locate important classes by looking how the system behaves at runtime.
- *How can we achieve the immediate availability of dynamic information in the IDE?* An important prerequisite to such an representation of collaboration in IDEs is an efficient and effective technique to gather dynamic information, which is fulfilled by sub-method partial behavioral reflection [DENK 07] as we illustrated by means of conducted benchmarks.
- *How do we browse and reason about dynamic collaborations in an IDE?* We identified several requirements an integration of dynamic collaboration in an IDE needs to fulfill in order to be useful to developers: (i) focusing on specific static artifacts (*e.g.* identified classes) is crucial, (ii) view generation needs to be immediately available after system execution and tightly integrated in the IDE, *i.e.* accessible while browsing static artifacts, (iii) these views of dynamic collaboration have to be condensed, interactive and intertwined with the static view on the system to be usable and to not force developers to switch to other tools or views separated from the conventional source views.

As we have shown in a preliminary user experiment with Smalltalk developers, the representation of dynamic collaboration as charts available in the IDE provides added value in terms of program comprehension, for instance feature comprehension, and navigation of static source artifacts.

After having presented three related approaches integrating dynamic information in the static source perspectives of IDEs, we are still missing a representation of features in the IDE. *Hermion* is able to improve the understanding of static source code and the execution flow therein, *Senseo* aims at integrating in IDEs information about how source artifacts (packages, classes, methods) collaborate to other artifacts, and *CollView* visualizes such collaboration information to provide an overview and a better means to navigate a system based on dynamic relationships between the various system artifacts. However, software features, which are dynamic by nature, are not explicitly represented by any of these three proposals. Thus, we elaborate in the following chapter on *FeatureEnv*, an enhancement to the IDE allowing developers to execute one or several features and visualizing the execution of the feature(s) in the development environment to better analyze, navigate, or maintain the feature(s).

Chapter 10

FeatureEnv – Visualizing Software Features in IDEs

10.1 Introduction

10.1.1 Positioning *FeatureEnv*

In this chapter we describe *FeatureEnv*, an enhancement to the conventional Squeak Smalltalk IDE. *FeatureEnv* sets out to tackle the invisibility of software features in the static source perspectives of IDEs by providing visualizations of features and by highlighting in the source views the artifacts contributing at runtime to a feature. *FeatureEnv* also mitigates the problem of hidden collaboration between conceptually related but statically distributed source artifacts by relating in the feature visualizations all source artifacts exercised during feature execution. Thus, *FeatureEnv* improves the support for all development activities except artifact usage investigation. Figure 10.1 summarizes the IDE problems tackled by *FeatureEnv* and the development activities for which it provides better support in the IDE.

To achieve its goals, *FeatureEnv* provides three perspectives on features integrated in the IDE: (i) a feature overview to compare the dynamics of several features to each other, (ii) a feature tree presenting the method call tree generated during the execution of a specific feature, and (iii) a feature artifact browser which is an adapted version of the traditional system browser in Smalltalk and which highlights all source artifacts (packages, classes, methods) used in a specific feature.

	Problem \ Activity	Feature investigation	Feature implement-ation	Artifact investigation	Dependency investigation	Runtime interaction investigation	Artifact usage investigation	Execution pattern investigation	Quality assessment	Domain concept analysis
Static views	Distributed artifacts	✔		✔	✔	✔		✔	✔	✔
	Collaboration hidden	✔		✔	✔	✔		✔	✔	✔
	Features hidden in code	✔	✔							✔

Figure 10.1: *FeatureEnv* addresses the problem of the invisibility of features in IDEs by explicitly representing them and also contributes to make visible hidden collaboration between distributed source artifacts.

The rest of this chapter is dedicated to presenting *FeatureEnv* in detail. After introducing the approach, we give reasons why a missing representation of features is a burden to program comprehension and software maintenance. We afterwards present *FeatureEnv*, which tackles this problem. We validate our proposal with a controlled empirical experiment concerned with typical software maintenance tasks.

10.1.2 Introduction to *FeatureEnv*

System comprehension is a prerequisite for software maintenance but it is a time-consuming activity. Studies show that 50-60% of software engineering effort is spent trying to understand source code [BASI 97, CORB 89]. Object-oriented language characteristics such as inheritance and polymorphism make it difficult to understand runtime behavior purely by inspecting source code [DEME 03, DUNS 00, WILD 92]. The task of understanding a software system is further exacerbated by a best practice in object-oriented programming to scatter behavior in many small methods, often in deep inheritance hierarchies [NIEL 89b].

The problems of understanding object-oriented software are poorly addressed by current development tools, since these tools typically focus only on a structural perspective of a software system by displaying static source artifacts such as packages, classes and methods. This is also true for modern Smalltalk dialects and development environments such as Squeak [INGA 97] or Cincom VisualWorks [VISU 10].

A system may be viewed as a set of features. Each feature represents a well-understood abstraction of a system's problem domain. Typical maintenance requests are expressed in terms of features [MEHT 02]. Thus, understanding how features are implemented is a prerequisite for system maintenance. A feature denotes a unit of behavior of a system. It exists at runtime as a collaboration of objects exchanging messages to achieve a specific goal. However, as it is not explicitly represented in the source

code, it is not easy to identify and manipulate. In this chapter, we adopt the definition of a feature as being a unit of observable behavior of a system [EISE 03].

As traditional development environments offer the software engineer a purely structural perspective of object-oriented software, they make no provision for the representation of behavioral entities such as features. To tackle this shortcoming, we propose to support the task of program understanding during maintenance by augmenting a static source code perspective with a feature perspective of a software system. We present a novel feature-centric environment called *FeatureEnv* which provides support for visual representation, interactive exploration, navigation and maintenance of a system's features. To motivate our work, we address the following questions:

- How useful is a feature-centric development environment for understanding and maintaining software?
- How do we quantitatively measure the usefulness of a feature-centric development environment?
- How do software engineers subjectively rate the usefulness of a feature-centric perspective of a system to perform their maintenance tasks?

The fundamental question we seek to answer is if software engineers can indeed better understand and maintain a software system by exploiting a feature-centric perspective in a dedicated feature-centric development environment. We want to determine if a feature-centric perspective is superior to a structural perspective to support program comprehension. To address this question, we implemented *FeatureEnv* in Squeak [INGA 97], a dialect of Smalltalk. *FeatureEnv* acts as a proof of concept for the technical feasibility of our approach and as a tool which we can validate in practice with software engineers. While this tool adds more information to the overloaded IDE, it at the same time also helps developers to better identify relevant artifacts and thus mitigates the negative impact of information overload.

The key contributions of this chapter are: (1) we present our feature-centric development environment, and (2) we provide empirical evidence to show its usefulness to support comprehension and maintenance activities as compared with the structural views provided by a traditional development environment.

Structure. In the next section we expand on the problem of feature comprehension and provide a motivating example. Based on this, we formulate our hypotheses of the usefulness of a feature-centric perspective

for performing maintenance tasks. In Section 10.3 we introduce *FeatureEnv*, allowing a developer to work in a feature-centric development environment. We validate the usefulness of *FeatureEnv* by conducting an empirical study in Section 10.4. We present the results and evidence of our study in Section 10.5. We report on related work in Section 10.6 and finally we conclude in Section 10.7.

10.2 Problem of Feature Identification

It is a generally accepted *best practice* of object-oriented programming that functionalities or *features* are implemented as a number of small methods [DUNS 00]. This, in addition to the added complexity of inheritance and polymorphism in object-oriented software, means that a software engineer often needs to browse a long chain of small methods to understand how a feature is implemented. In Figure 10.2 we illustrate this with an example from Pier [RENG 06], the system we chose as a basis for our experimentation. Pier is a web content management system [RENG 06] implemented in Squeak Smalltalk [INGA 97]. Figure 10.2 shows a small part of the class hierarchy of Pier and an excerpt of a call tree, generated by exercising the *copy page* feature.

A software engineer, faced with the task of maintaining the *copy page* feature, first needs to locate the relevant classes and methods, and then browses back and forth in the call chain to establish a mental map of the relevant parts of the code and to gain an understanding of how the feature is implemented. This is a cumbersome and time-consuming activity and often the software engineer loses time and focus while browsing irrelevant code.

10.2.1 Explicitly Representing Features in the IDE

From the perspective of a software engineer, a feature consists of a set of all methods executed while performing a certain task or activity in a software system. The relationships between the methods of a feature are dynamic in nature and thus are not explicit in the structural representation of the software [JERD 96]. For our purposes, we represent features (*i.e.* dynamic units of behavior) in terms of their participating methods. We aim at supporting software maintenance activities in general (as discussed in Section 10.1.2), though the main focus of *FeatureEnv* is on supporting the software engineer when maintaining or fixing a defect in a feature. Thus we need to capture and represent features as explicit entities. The behavior of a feature may be captured by triggering an activity from the user

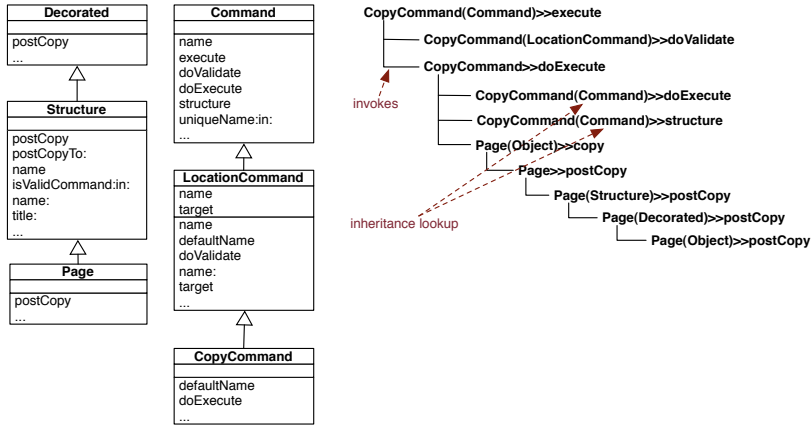

Figure 10.2: The relevant Pier class hierarchies for the *copy page* feature and its call graph.

interface. Alternatively, as described in the work of Licata *et al.* [LICA 03], test cases are typically aligned with features. For our experimentation we opted to use test cases to trigger features. Furthermore, by using test cases we can better control the volume of dynamic information captured to represent each feature.

Our premise is that by explicitly representing features in the development environment, we support maintenance activities by providing the software engineer with an explicit map between features and source entities that implement the feature. By focusing the attention of the maintainer on only relevant source entities of a given feature or a set of features, we improve the understanding and the ease with which she can carry out maintenance tasks. This clear focus may actually reduce the negative impact of information overload even though *FeatureEnv* integrates more information in the IDE; as developers have to spend less time to locate artifacts relevant for a particular feature, they are less hampered by overloaded interfaces in the IDE.

We state our hypotheses as:

- *FeatureEnv decreases the time a software engineer has to spend to maintain a software system* (e.g. *to correct a bug) compared to a traditional development environment which provides only a structural perspective of the code.*

- *FeatureEnv improves and enriches the understanding of how the features of a software system are implemented.*

We refine our hypotheses in Section 10.4, when we describe the details of our empirical study. Our qualitative and quantitative evaluation of the findings of our experimentation reveal that these hypotheses indeed hold.

10.3 *FeatureEnv*, a Feature-centric Environment

We embed *FeatureEnv* in the software engineer's integrated development environment (IDE). The purpose of *FeatureEnv* is to augment an IDE with a feature perspective of a software. We implemented our approach in Squeak Smalltalk [INGA 97]. *FeatureEnv* complements the traditional structural and purely textual representation of source code in a browser by presenting the developer with interactive, navigable visualizations of features in three distinct but complementary views. These views are enriched with metrics to provide the software engineer with additional information about the relevancy of source artifacts (*i.e.* classes and methods) to features.

Initially, we introduce the key elements of *FeatureEnv*. Subsequently, we describe how *FeatureEnv* promotes a software engineer's comprehension of scattered code in object-oriented programming while performing maintenance tasks on a system's features.

10.3.1 Feature Affinity in a Nutshell

Greevy *et al.* [GREE 07] defined a *Feature Affinity* measure to assign a relevancy scale to methods in the context of a set of features. Feature Affinity defines an ordinal scale corresponding to increasing levels of participation of a source artifact (*e.g.* a method) in the set of features that have been exercised. For *FeatureEnv* we consider four Feature Affinity values: (1) a *singleFeature* method participates in only one feature, (2) a *lowGroupFeature* method participates in less than 50% of the features, (3) a *highGroupFeature* method participates in 50% or more of the features and (4) an *infrastructuralFeature* method participates in all of the features.

We exploit the semantics of *Feature Affinity* to guide and support the software engineer during the navigation and understanding of one or many features. We assign to the visual representation of a method a color that represents its *Feature Affinity* value. Our choice of colors corresponds to a heat map (that is, a cyan method implies *singleFeature*

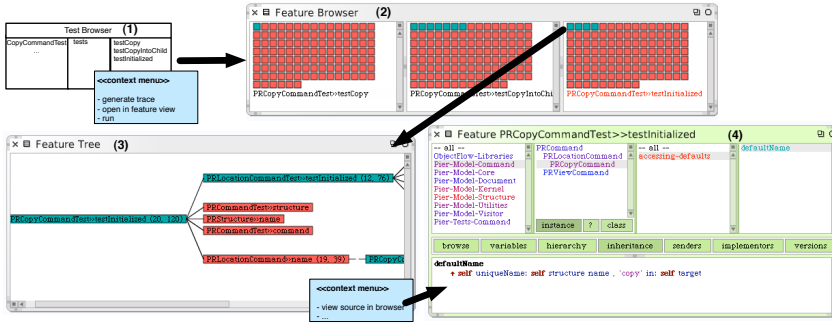

Figure 10.3: The Elements of our *FeatureEnv*.

and red implies *infrastructuralFeature*, *i.e.* used by all the features we are currently investigating). Such a heat map allows developers to quickly grasp the similarities and differences between two or more features. Such knowledge helps developers to better understand features and to locate artifacts responsible for flaws in particular features. It is likely that a defect occurring in one single feature is caused by parts that are different from similar but non-broken features.

10.3.2 Elements of *FeatureEnv*

FeatureEnv contributes three different visualizations for one and the same feature: (2) the compact feature overview, (3) the feature tree view and (4) the feature artifact browser. (1) is the test runner which is not directly part of *FeatureEnv* but a separate tool. Any means exercising a feature of a software system can be used by *FeatureEnv* to analyze this feature. In Figure 10.3 we use a test runner.

Compact Feature Overview

The Compact Feature Overview presents a visualization of two or more features represented in a compacted form. The Compact Feature view represents a feature as a collection of all methods used in the feature as a result of capturing its execution trace. Each method is displayed as a small colored box; the color represents the *Feature Affinity* value. The methods are sorted according to their *Feature Affinity* value. The software engineer decides how many features she wants to visualize at the same time (see Figure 10.3 (2)). Clicking on a method box in the Compact Feature View opens the Feature Tree View, which depicts a call tree of the execution trace. This visualization reveals the method names

and order of execution. All occurrences of the method selected in the Compact Feature View are highlighted in the call tree.

Feature Tree

This view presents the method call tree, captured as a result of exercising one feature (see Figure 10.3 (3)). The first method executed for a feature (*e.g.* the "main" method) forms the root of this tree. Methods invoked in this root node form the first level of the tree, hence the nodes represent methods and the edges are message sends from a sender to a receiver. As with the Compact Feature Overview, the nodes of the tree are colored according to their *Feature Affinity* value.

The key challenge of dynamic analysis is how to deal with the large amount of data. For our experimentation we chose to analyze Pier [RENG 06], a web-based content management system implemented in Smalltalk. We obtained large traces containing more than 15'000 method invocations. We discovered that it is nearly impossible to visualize that amount of data without losing the overview and focus while still conveying useful information. To overcome this we applied two techniques:

First, we opted to execute test cases of a software system rather than interactively trigger the features directly from the user interface. For instance, instead of looking at the entire *copy page* feature initiated by a user action in the user interface, we analyze the *copy page* feature by running the test cases that were implemented to test this feature. As stated in the work of Licata [LICA 03], features are often encoded in dedicated unit test cases or in functional tests encoded within several unit test cases. In the case of a software system that includes a comprehensive test suite, it is appropriate to interpret the execution traces of such test cases as feature execution traces [LICA 03].

Second, we compressed the execution traces and the corresponding visual representation as a feature tree as much as possible without omitting information about order of execution of method sends. To achieve this we use two different algorithms, one to remove common subexpressions in the tree and another to remove sequences of recurring method calls as a result of loops. These two algorithms are explained in the following.

Common Subexpression Removal

A subexpression in a tree is a branch which occurs more than once. If, for example, a pattern "method c invokes methods a and b" occurs several times in a call tree, we identify this pattern as a common subexpression. Our analysis reveals that the execution traces

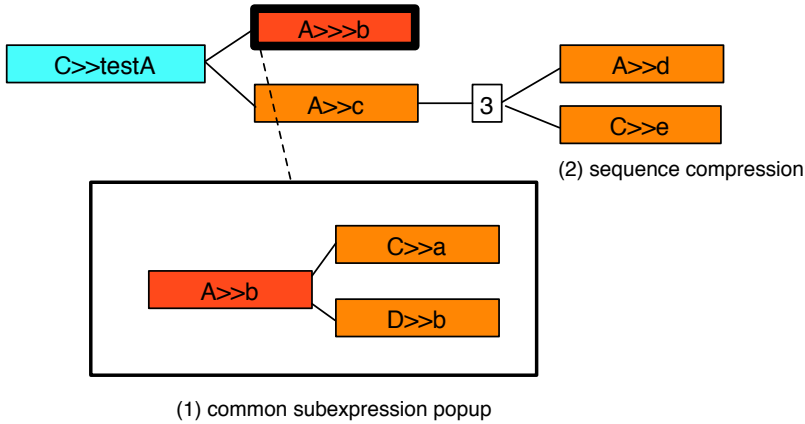

Figure 10.4: The Common Subexpression and Sequence Compression of the Feature Tree.

of features typically contain many common subexpressions. By compacting the representation of these subexpressions, we reduce the tree by up to 30% on average. Our visualization still includes an expandable root node of a common subexpression branch in the tree, the subexpression can be opened in a pop-up window by the software engineer. Figure 10.4 (1) shows a schematic representation of how we display common subexpressions in our feature tree view.

To remove common subexpressions, we applied the algorithm presented by Flajolet *et al.* [FLAJ 90].

Sequence Removal

Often a feature trace contains several sequences, *e.g.* methods invoked in a loop. It is straightforward to compress these nodes included in a loop by only presenting them once. Furthermore, we indicate how often the statements of a loop are repeated. If, for example, the methods d and e are executed three times in a loop, we add an *artificial numeric* node labeled with a '3' to the tree and link the nodes for d and e to this node (as shown in Figure 10.4 (2)). To detect and compact sequences, we implemented a variation of the algorithm presented in Hamou-Lhadj *et al.* [HAMO 03].

These techniques guarantee that the feature tree is still complete and easy to read and interpret by the software engineer. No method calls occurring in the feature are omitted.

Initially a feature tree is displayed collapsed to the first two levels. Every node can be expanded and collapsed again. In this way, the software engineer can conveniently navigate even large feature trees.

When a user selects a method node in the compact feature overview, all the occurrences of this method are highlighted in the feature tree. Also the tree is automatically expanded to the first occurrence of the selected method and the feature tree window is centered on that node. The user can navigate through all occurrences of the desired method by repeatedly clicking on the corresponding node in the compact feature overview. By opening a method node in the feature tree the engineer is able to follow the complete chain of method calls from the root node to the current opened method node. No intermediate calls of methods belonging to the software system under study are omitted.

Every node of the tree provides a button to view the source code of the corresponding method in the feature artifact browser.

Feature Artifact Browser

The source artifacts of an individual feature are presented as text in the *feature artifact browser* (see Figure 10.3 (4)). By default, it exclusively displays the classes and methods actually used in the feature. On demand, developers can also see all system artifacts in the feature artifact browser; the feature artifacts appear highlighted. This makes it much easier for the user to focus on a single feature of the software. Our *feature artifact browser* is an adapted version of a standard class browser available in the Squeak environment. It contains packages, classes, method categories and methods in four panes on the top, while the lower pane contains the source code of the selected entity. This version of the class browser not only presents static source artifacts, but also the feature affinity metric values by coloring the names of classes and methods accordingly.

The three distinct visualizations provided by *FeatureEnv* are tightly interconnected so that a software engineer does not lose the overview when performing a maintenance task. For instance, the user selects a method in the compact feature overview, the tree opens with all occurrences of the selected method. From the tree perspective, the software engineer can choose to view a method as source code in the feature artifact browser by double-clicking on a node that represents the method of interest in the feature tree.

To initiate a maintenance session with *FeatureEnv*, the software engineer selects test cases that exercise features relevant to her specific maintenance task. To ease the collection of features traces, we extended the standard class browser of Smalltalk (called OmniBrowser) to seamlessly execute instrumented test cases and capture feature traces as shown in Figure 10.3 (1). Thus, once she has executed the instrumented test cases, the software engineer launches *FeatureEnv* with an initial number of features.

10.3.3 Maintaining Software with *FeatureEnv*

FeatureEnv supports program comprehension and feature understanding for maintenance tasks in three ways:

Firstly, by comparing several features with each other in the compact feature overview, software engineers gain knowledge about how different selected features are related in terms of the methods they have in common. Since the features included in this compact feature overview can be arbitrarily selected, a developer can compare any two features with each other. However, when performing a specific maintenance task, it makes sense to select related test cases to *e.g.* compare failing tests with similar, non-failing tests to determine which parts of a software system may be most likely responsible for a defect. If, for example, only one test method of a test class exercising the *copy command* feature of a software system is failing, then we compare this test method to all test methods exercising the *copy command* feature. We assume that by looking first at the methods that are only used in the single failing test method, we are more likely to be faster at discovering the defect, as chances are high that one of the *singleFeature* methods (*i.e.* methods unique to one compact feature view) is responsible for that defect. The aim of the compact feature overview is to support the quick identification and rejection of candidate methods that may contain a defect.

Secondly, the feature tree provides insight into the dynamic structure of a feature, an orthogonal dimension compared to the static structure visible in the source code. The nodes in the tree are also colored according to the Feature Affinity metric which guides the software engineer to identify faulty methods. In the example described above where a single feature is failing, the most likely candidate methods responsible for the defect are colored in cyan in the feature tree. Since this tree is complete, *i.e.* it contains all method calls for that specific feature, chances are high that the engineer discovers the source of the defect in one of these single methods that are easy to locate in the feature tree due to their coloring.

The software engineer can also navigate and browse the tree to obtain a deeper understanding of the implementation of the feature and the relationships between the different methods used in the feature. For every method of a feature, the software engineer can easily navigate to all occurrences of this method in the feature tree to find out how and in which context the given method is used. This helps one to discover the location of a defect and the reason why it occurs. The feature tree transforms and improves the understanding of the dynamic structure of a feature and reveals where and how methods are used. For every node in the feature tree, the developer can view the source code of that method in the feature artifact browser.

Thirdly, the feature artifact browser helps the developer to focus only on entities effectively used in a feature. The number of methods that might be responsible for bugs is thus reduced to a small subset of all existing methods in a class or a package. Since only packages, classes and methods are presented to the developer in the feature artifact browser, it is much easier for her to find information relevant for a defect or another maintenance task, *i.e.* classes or methods. Hence the feature artifact browser helps one to focus on relevant source artifacts and to not lose track and context and thus, in a sense, reduces the information overload by allowing developers to focus on a subset of all system entities.

10.4 Validation

To obtain a measure of the usefulness of *FeatureEnv* and its concepts in practice, we conducted an empirical study with subjects using and working with *FeatureEnv*. The goals of this study are to gain insight into the strengths and shortcomings of our current implementation of *FeatureEnv*, to obtain user feedback about possible improvements and enhancements, and to assess the practical potential of *FeatureEnv*. Our primary goal is to gather quantitative data that indicates how beneficial is the effect of using *FeatureEnv* as compared with the standard structural and textual representations of a traditional development environment. We introduce and describe the experiment in this section, formulate the hypotheses we address and describe precisely the study design. Finally, we present the results we obtained from the experiment.

10.4.1 Introducing the Experiment

To validate *FeatureEnv* we asked twelve subjects (computer science graduate students) to perform two equally complex maintenance tasks in a

$H0_1$	The time to discover the location of the defect is equal when using the standard browser and *FeatureEnv*. (formally: $\mu_{D,FB}$ = $\mu_{D,OB}$, where $\mu_{D,FB}$ is the average discovery time using *FeatureEnv* and $\mu_{D,OB}$ the average discovery time using OmniBrowser)
$H0_2$	The time to correct the defect is equal when using the standard browser and *FeatureEnv*. (formally: $\mu_{C,FB} = \mu_{C,OB}$)
$H0_3$	*FeatureEnv* has no effect on the software engineer's program comprehension. (formally: average effect $\mu_{E,FB} = 0$)

Table 10.1: Formulation of the null hypotheses.

software system, one task performed in *FeatureEnv* and the other in the standard environment of Squeak Smalltalk (*i.e.* using OmniBrowser). As a maintenance task, we assigned the subjects the correction of a defect in the software system. The presence of the defect is revealed by the fact that some of the feature tests are failing.

In this experiment we seek to validate three hypotheses concerning *FeatureEnv*. If the result of the experiment reveals that the hypotheses hold, we then have successfully obtained clear evidence that *FeatureEnv* supports a developer to perform maintenance tasks and that the feature affinity metric we applied is of value in practice.

10.4.2 Hypotheses

We propose the three null hypotheses listed in Table 10.1. The goal of the experiment is to refute these null hypotheses in favor of alternative hypotheses which would indicate that a feature-centric environment helps the software engineer to discover and correct a defect and hence improves program comprehension.

10.4.3 Study design

Study setup. During the experiment, subjects were asked to correct two bugs in a complex web-based content management framework written in Smalltalk. Our software system, Pier [RENG 06], consists of 219 classes of 2341 methods with a total of 19261 lines of code. The two defects are approximatively equally complex to discover and correct. To introduce these two bugs we slightly changed one method per bug in the Pier system. As a result of our change, some (feature) tests failed. We presented the subjects with these failing tests as a starting point for their search for

the defect. In Pier a unit test class is dedicated to a certain feature (*e.g.* copying a Wiki page) and the different methods of a test class are different instantiations of that feature (*e.g.* different parameters with which the feature is exercised). This is in line with the argumentation presented in the work of Licata [LICA 03] stating that features are often encoded in unit test cases.

In our experiment, we introduced two different defects in the *copy page* feature. This feature is tested by a dedicated test class with five test methods. The two defects produce failures in different test methods of the *copy page* test class. For the experiment we select all five test methods exercising the *copy page feature* and show them in row in the Compact Feature View of *FeatureEnv*. As these five test methods exercise variants of the same feature, they are clearly related to each other, which means that if one test method shows a failure but the others do not, the failure is most likely caused by methods the failing test is exclusively executing.

We conducted the experiment with twelve graduate computer science students as subjects with varying degrees of experience with the Smalltalk programming language and the Squeak development environment. All subjects had between one and five years of experience with the language, but only between zero and four years of experience with the Squeak development environment. None of the subjects was familiar with the design and implementation of the Pier application in detail.

Before starting the experiment, we organized a workshop to introduce the concepts and paradigms of *FeatureEnv*. Every subject could experiment with *FeatureEnv* for half an hour before commencing our experiment consisting of the task of defect location and correction. Furthermore, we briefly introduced the subjects to the design and the basic concepts of Pier by presenting an UML diagram of the important entities of the application. The experiment was conducted in a laboratory environment, as opposed to the subjects' normal working environment. While performing the experiment, we observed the subjects. Afterwards we asked them to respond to a questionnaire to gather qualitative information about *FeatureEnv*. The questionnaire contains several questions about the usefulness of *FeatureEnv* to understand the program and to perform the requested maintenance tasks. For every question, the subjects could choose a rating from -3 to 3, where -3 represents a hindrance to program comprehension, 0 no effect and 3 very useful. In addition, the subjects could provide qualitative feedback, *e.g.* identified shortcomings of the environment and suggestions for improvement. The results of these open suggestions, as well as the observations of the experimenters form the qualitative part of our study.

Every subject had to fix both defects, one using *FeatureEnv* and the other one using the standard class browser, *i.e.* OmniBrowser. Both the debugger as well as the unit-test runner were available to complete the task. We prohibited use of every other tool during the experiment. From subject to subject we varied the order in which they fixed the defects as well as the order in which they used the different browsers. Hence there are four possible combinations to conduct an experiment with a subject and each of these four combinations was exercised three times. A concrete combination was randomly chosen by the experimenters for any subject.

Dependent variables. We recorded two dependent variables: (i) the time to discover the location (*i.e.* the method) where the defect was introduced, and (ii) the time to actually correct the defect completely. We considered the goal as being achieved when all 872 unit tests of Pier ran successfully. The bugs were carefully chosen so that they could only be corrected in a specific method.

10.4.4 Study Result

Initially we report on the quantitative data we obtained by recording the time the subjects spent to locate and correct the defects using the different browsers. Then we present qualitative feedback by evaluating the results from the questionnaire answered by the subjects.

Time Evaluation. Figure 10.5 compares the average time spent to correct the bugs, independently of which browser has been used to fix a defect. The figure clearly shows that the two bugs were approximately equally complex, which allows us to compare the time different subjects spent in different environments to correct the two defects. We initially selected these two defects after having assessed their complexity in a pre-test with two subjects working in the standard Squeak browser. These two subjects are not included in the final evaluation of the experiment. They needed approximately the same time to correct both defects and subjectively considered the two bugs as equally complex and difficult to correct.

Figure 10.6 compares the total time the subjects spent to discover the location of the defect once using *FeatureEnv* and once using the OMNIBROWSER. *FeatureEnv* yields a 56 percent decrease in time spent which

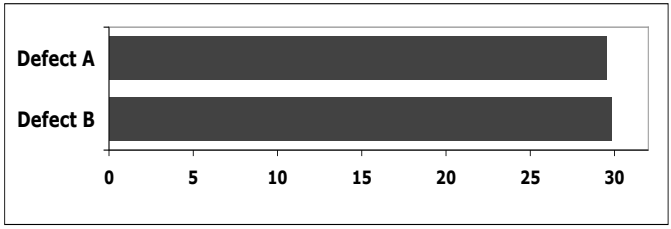

Figure 10.5: Comparing average time to correct the two defects.

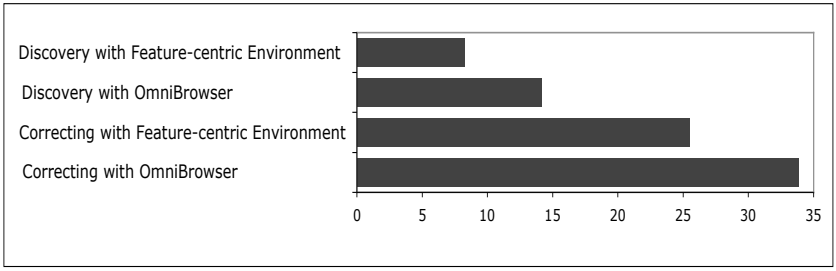

Figure 10.6: Comparing average time between using *FeatureEnv* and OMNIBROWSER to discover and correct a defect.

equals to 56 minutes saved compared to using OMNIBROWSER. During the experiment we considered that the correct location of the defect has been discovered when the subject announces to the experimenters the method considered to be responsible for the defect. If the announced method was incorrectly blamed, the experimenters did not accept the answer and subjects continued searching.

The situation is similar when considering the time spent to fully correct the defects (see also Figure 10.6), which is the time to discover the defect plus the amount of time to edit and correct the faulty method. Here we get a relative improvement of 33 percent which equals 100 minutes saved when using *FeatureEnv* instead of OMNIBROWSER. Figure 10.7 presents boxplots showing the distribution of the discovery and correction time the different subjects spent in different browsers. The different defects are not identifiable in these boxplots, only the different browsers. To determine the complete correcting time we considered the time spent to make all tests of Pier run successfully.

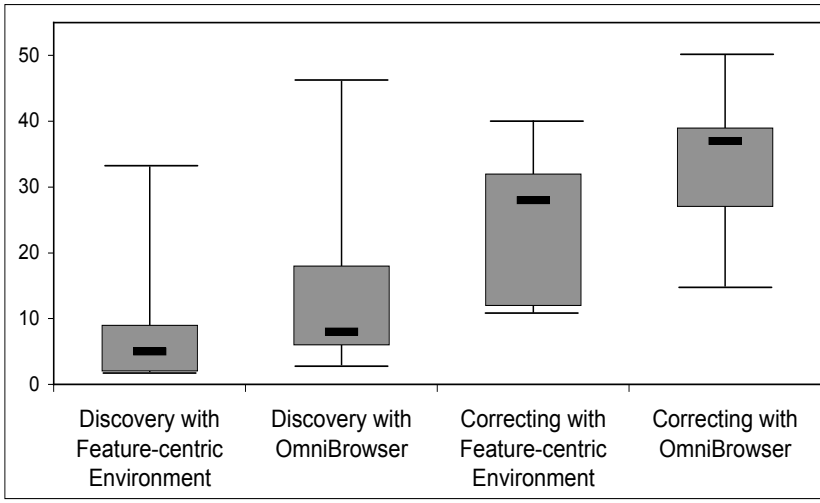

Figure 10.7: Boxplots showing the distribution of the different subjects.

Qualitative Feedback. In our questionnaire we mainly asked the subjects how they rated the effect of the different aspects of *FeatureEnv* on program comprehension. We asked about the overall effect of *FeatureEnv* on program comprehension and about the specific effect of feature overview, feature tree, feature class browser, and the feature affinity metric. Furthermore, we asked how well certain parts of *FeatureEnv* were understood by subjects and how well they could interact with the different parts. In Table 10.2 we present the details of our questionnaire and the average results we got from the subjects. They could choose between a rating from -3 to 3, so an average rating of 1.16 for *e.g.* "General effect on Program Comprehension" reveals a positive, although not a very strong effect. As an example, we depict in Figure 10.8 the results for the question about the effect of the compact feature overview on program comprehension. The ratings were on average 1.58 which denotes that the subjects considered the effect on average as "good".

Statistical Conclusion. To test the first two hypotheses formulated in Table 10.1 we apply the one-sided independent t-test [KANJ 99] with an α value of 10% and 22 degrees of freedom. One requirement for applying the t-test is equality of variance of the two samples. For the two discovery time samples we determine a variance of 92 and 112, respectively; for

	Question	Result
1.	General effect on Program Comprehension	1.16
2.	Effect of Compact Feature Overview	1.58
3.	Effect of Feature Tree Browser	1.33
4.	Effect of Feature Class Browser	1.50
5.	Effect of Feature Affinity Metric	1.42
6.	Understanding the subexpression compression in Feature Tree	1.58
7.	Understanding the sequence compression in Feature Tree	1.75
8.	Understanding the navigation in Feature Tree	2.00
9.	Interaction with Compact Feature View	1.50
10.	Interaction between Feature Tree and Feature Class Browser	1.58

Table 10.2: Questionnaire.

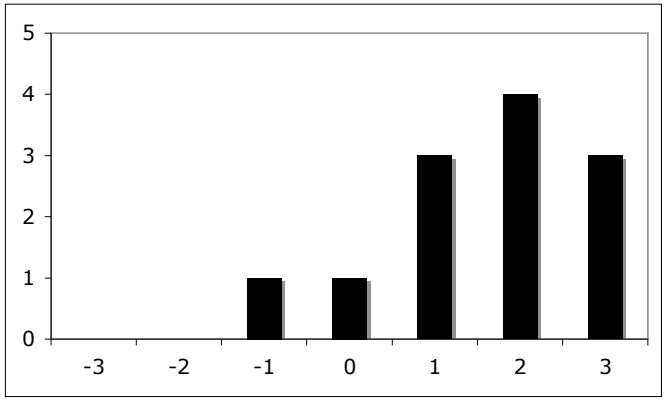

Figure 10.8: Comparing the average results for the effect of compact feature overview on program comprehension.

the correcting time samples the variances are closer to each other, 102 and 110, respectively. For the correction time the variance requirement is fulfilled. For the discovery time we are careful and assume that this requirement is not fulfilled. Another requirement for applying the t-test is a normal distribution of the underlying data which we justify with the Kolmogorow-Smirnow-Test. With an α value of 5% the result of this test allows us to assume normal distribution for the correction time samples. With an α value of 10% we can also assume normal distribution of the discovery time samples.

These preliminary tests allow us to use the t-test at least for the correction time. We also use it for the discovery time, but we are skeptical about the outcome. For the discovery time we calculated a t value of 1.32. The

t distribution says that the probability of t $>$ 1.32 is exactly 10% which means that we can just barely reject the null hypothesis $H0_1$. Because the requirements for the t-test are not properly fulfilled and because we consider an α value of 10% to be too low to justify a rejection of the null hypothesis, we cannot give enough evidence for the positive effect of *FeatureEnv* on the discovery time of a defect.

For the time to correct a defect we obtain a t value of 1.86 which is bigger than the t value of the 95% confidence interval (t = 1.717). This means that we can reject $H0_2$ with an α value of 5% and accept the alternative hypothesis $H1_2$ saying that *FeatureEnv* speeds up the time to correct a defect ($\mu_{C,FB} < \mu_{C,OB}$).

At first glance it looks surprising that we cannot reject $H0_1$ but $H0_2$, which means that the experiment substantiates our claim that *FeatureEnv* helps developers correcting defects more efficiently, but not that *FeatureEnv* also improves locating the artifacts causing these defects. We explain this result by the fact that there is a significant outlier in the discovery time data with the feature-centric environment (cf. Figure 10.7). One subject spent a considerable amount of time locating the appropriate method to correct the defect while the actual correction just took little time. We did not remove this outlier from the evaluation as it was not caused by an abnormal condition that would justify such a removal. As we only had twelve subjects available for this experiment, outliers can have a huge impact on the results as there are not enough data points that could compensate for such outliers. We are confident that we could also reject $H0_1$ when repeating this experiment with considerably more subjects than just twelve, in particular as we were still able to reject $H0_2$ even with the present experiment.

To test the third hypothesis we use the results of the questionnaire and apply the one-sided Wilcoxon signed-rank test [WILC 45]. We cannot assume normal distribution for the underlying data in this case since the ratings were almost all positive, thus we opted for applying the Wilcoxon test. We only apply the test to the answers for the general effect of *FeatureEnv*. We calculate a W value of 26 which is exactly equal to the S value we find in the tabular denoting the 95% confidence interval. This means that we can reject $H0_3$ with an α value of 5% and hence accept the alternative hypothesis which says that *FeatureEnv* has a positive effect on program comprehension.

10.4.5 Threats to Validity

In this section we report on the main threats to validity of our experiment. We distinguish between external, internal and construct validity [O'Br 05].

- *External validity* depends on the subjects and the software system in which subjects are asked to correct defects during the experiment. In our case the software system, Pier, is a complex real-world application comparable to other industrial object-oriented systems. The subjects, however, are students and research assistants who may not be directly comparable to programmers in industry. Furthermore, the experiment was conducted in a laboratory environment which biases the performance of the subjects. Hence the results are not directly applicable to settings in practice, although we consider the aforementioned influences to be small.

- *Internal validity* is jeopardized by the fact that not all subjects had the same amount of experience with the programming language and especially with the Squeak development environment. While everybody was quite familiar with programming in Smalltalk (at least one year of experience), the particular environment was new to some of the subjects. But because these problems were present for both defects and browsers, we believe that they do not bias the results tremendously. Furthermore, we changed the order of the bugs and browsers from subject to subject, hence the results were only slightly biased by the fact that subjects had more insight into the development environment and also into the software system when fixing the second bug than they had for the first bug. However, the different amount of experience of the subjects is nonetheless a shortcoming of this study.

  Another important issue is that subjects could also use other tools than just the two different browsers, *e.g.* the debugger. As it is usually necessary to use a debugger in a dynamic language to find a bug, we did not prohibit its usage. Some subjects performed the task predominantly by using the debugger to find the bug whereas they made the necessary corrections often in the provided browser. Using the debugger was a more frequent phenomena when subjects used the traditional browser to fix the defect. These other available tools clearly bias the result of our experiment to a degree which is hard to estimate.

  Yet another important issue is that the subjects were unfamiliar with *FeatureEnv*. It is a complete new environment with a very

different approach to look at a software system and its features. The other environment (*i.e.* the OmniBrowser) was well-known by all subjects, since the paradigm applied in this browser is the standard way of browsing source code in most Smalltalk dialects.

- *Construct validity*. Our measure, the time to find and correct a defect, is adequate to assess the contribution of *FeatureEnv* to maintenance performance. However, this time is certainly biased by many other factors than the browsers in use, such as the experience of the developer, her motivation, the use of other tools, etc. To assess the effect of *FeatureEnv* on program comprehension we used our questionnaire to obtain feedback on how the subjects personally judge the effects on program comprehension. These answers are certainly subjective and may hence not be representative. Thus, the applied measures are not a perfect assessment of the effects of *FeatureEnv* on software maintenance, although at least the former is still relatively well assessed with the applied measures.

10.4.6 Study Conclusion

Two main issues of our empirical study are: (1) the subjects participating in the experiment do not have the same experience with the programming language and the development environment, and (2) they were unfamiliar with *FeatureEnv* as they had never used it before. However, they are familiar with the standard development environment. Another important issue is that due to the limited number of subjects participating in the experiment, it is difficult to draw statistically firm conclusions. This study nonetheless gives us worthwhile insights into how software engineers use *FeatureEnv* and how they judge its impact on software maintenance. The results we obtained motivate us to proceed with our work on this environment and the subject feedback gives us ideas for its improvement. We conclude that performing this study was crucial to validate and improve *FeatureEnv*.

10.5 Discussion

In this section we discuss some of the interesting ideas and suggestions we obtained as feedback from the subjects who participated in the experiment. Furthermore, we also discuss some of the issues of feature analysis inherent in *FeatureEnv*:

- *Bidirectional interactions.* Providing the capability to navigate the tree by clicking and selecting the textual representations of the methods in the feature artifact browser and being able to click on nodes in the tree to select the same methods in the compact feature view is a useful improvement to *FeatureEnv*. This helps one to navigate and understand more quickly the structure and implementation of a feature.

- *Bind tree to debugger.* Using a debugger is not an easy task since we only see a slice of a program in the debugger but not the overall structure. If we could use the feature tree to step through a running program to debug it, we could easier gain an overview and understanding of the overall structure of a feature. Hence a promising extension of *FeatureEnv* would be to add debugging facilities to the feature tree, such as stepping through a program, inspecting variables and changing methods on the fly.

- *Delta debugging.* Using *FeatureEnv* to discover and correct a defect in a feature is a frequent task which we can ease by analyzing test cases representing features. Careful analysis of test cases with delta debugging approaches [ZELL 02] allows us to rank methods used in a feature according to their probability being responsible for a defect. If we present the methods in the compact feature view sorted by their probability to contain erroneous code, a developer can very quickly focus on the right methods to correct a defect.

- *Performance analysis.* By enriching *FeatureEnv*, in particular the feature tree, with more dynamic information such as execution time or memory consumption, the tool will be well suited for performance analysis. The feature tree can easily reveal which branch consumes the most resources or which specific method call takes the most time to execute. Another useful enhancement is a mechanism to compare different executions of a feature (*e.g.* with different parameters) with each other to emphasize differences in execution time or consumption of resources.

- *Scalability issues.* Dynamic analysis approaches are required to manipulate large amounts of data. We address this issue by using unit test cases to trigger the behavior of features. Test are usually constrained units of behavior, thus generate smaller amount of data than actual software features. Furthermore, we present the user with both a compressed feature view as well as with the entire call tree of a feature. To reduce the call tree, we applied compression algorithms. Our feature representations could also be

reduced by applying selective instrumentation and filtering techniques [HAMO 03].

- *Coverage.* By using test cases to represent features we do not obtain full coverage of all possible execution paths of a feature. Other feature identification approaches are, however, also subject to this limitation [WILD 95, EISE 03]. We argue that although full coverage is desirable, it is not essential to support a feature-centric approach to software maintenance.

10.6 Related Work

In this section we relate *FeatureEnv* to similar work in the field of software visualization and reverse engineering. We particularly compare our approach to proposals visualizing the dynamics of object-oriented systems. Most proposals visualize entire system executions and do not isolate specific software features in this execution. These related proposals are Whorf [BRAD 92], Program Explorer [LANG 95], Jinsight [DE P 93], GraphTrace [KLEY 88], and Jive/Jove [REIS 03, REIS 05]. All these approaches make use of dynamic (trace-based) information.

Whorf [BRAD 92] provides explicit support for delocalized plans (conceptually related code that is not located contiguously in a program). Whorf links different views on the software to highlight interactions between physically disparate components. The authors also performed an experiment with software engineers to measure how quickly it took them to identify relevant code to perform an enhancement to a software system, once with paper documentation and once with Whorf. The results show that using Whorf improved efficiency when performing a maintenance task. As with our experiment, the authors of Whorf were also able to obtain evidence that analyzing the interactions between distributed source artifacts and linking these artifacts together in a tool helps developers during software maintenance.

Program Explorer [LANG 95] focuses on visualizing design patterns to better navigate and understand frameworks. It does not visualize software features per se, but provides visualizations that scale well even for large applications by presenting abstracted dynamic information combined with results from static analysis. However, the dynamic information presented is not complete and not integrated in the IDE, thus this

approach is less useful during feature maintenance, particularly defect correction.

Jinsight [LANG 95] visualizes interactions between objects in interaction diagrams. These diagrams are mainly used to detect performance bottlenecks, thus the support of Jinsight for feature comprehension and software maintenance is rather limited.

GraphTrace [KLEY 88] uses graphs to visualize the behavior of object-oriented systems. GraphTrace shows the current program activity by highlighting in the graph the nodes (source artifacts) and edges (method invocations) currently being executed. As these graphs usually grow very large, their use during software maintenance is limited. As GraphTrace does not explicitly represent specific software features in a the graph, this tool does not help developers much during feature comprehension and maintenance.

Jive and Jove [REIS 03, REIS 05] visualize the runtime activity of Java programs in real time. Representing features or connections between disparate artifacts is not an objective of these tools, instead they aim at supporting debugging or performance optimizations. Thus, they contribute to rather specific tasks than to general software maintenance activities such as feature comprehension.

In contrast to the above approaches, *FeatureEnv* aims at directly incorporating interactive and navigable visualizations of the dynamic behavior of features into the development environment. In this way, we emphasize the importance of providing the software engineer with direct access to the information during a maintenance session.

10.7 Summary of the Chapter

In this chapter we presented *FeatureEnv* which allows us (1) to visually compare several features of a software system, (2) to visually analyze the dynamic structure of a single feature in detail and, (3) to navigate, browse and modify the source artifacts of a single feature in a feature artifact browser focusing on the entities actually used in that feature. All these visualizations are enriched with the feature affinity metric to highlight parts of a feature relevant to a specific maintenance task.

The views on features are fully interactive and interconnected to ease and enhance their usage in maintenance activities. We validated *Fea-*

tureEnv by carrying out an empirical study with twelve graduate computer science students. The results of our experiment are promising because they clearly reveal that *FeatureEnv* has a positive effect on software maintenance, in particular on the efficiency of correcting software defects. We recognize that, as our experiment had only a low number of participating subjects, it is difficult to generalize the results. However, feedback of the users in addition to the quantitative results of our analysis are encouraging.

With *FeatureEnv* we eventually contributed an approach representing software features in the IDE. With this proposal we complete the integration of dynamic information in the IDE after having presented approaches enriching the static source perspectives with behavioral information (*Hermion*, *Senseo*), linking distributed but dynamically interconnected artifacts (*Hermion*, *Senseo*), and visually representing dynamic collaboration patterns between such artifacts (*Senseo*, *CollView*). We conclude in the next chapter this second part of the dissertation by critically discussing our achievements.

Chapter 11

Discussion

In this chapter, we discuss the different approaches presented in the second part of the dissertation. We analyze which problems the various proposals combined address to which degree. We also consider the approaches introduced in the first part of the dissertation to obtain a complete overview of how we tackled the various IDE problems as identified in Section 1.1.2 in this dissertation. This analysis eventually reveals which problems are not yet fully addressed.

11.1 Problems Addressed in the Second Part

Figure 11.1 summarizes all problems addressed in the second part of the dissertation and reports on the development activities that are now better supported.

Hidden collaboration between distributed artifacts. All four approaches (*Hermion*, *Senseo*, *CollView*, and *FeatureEnv*) tackle the problem that conventional IDEs do not make visible dynamic collaboration between distributed source artifacts. Usually, IDEs have a narrow focus on static relationships between source artifacts (inheritance, package/-class/method relations, etc.), thus the dynamic communication between artifacts spread over the entire software space is difficult to discover in IDEs. Both *Hermion* and *Senseo* tackle this problem by embedding in the source code views dynamic information to link artifacts based on how they are dynamically used. *Senseo* additionally provides a navigable view showing callee and caller relationships between source artifacts, for

Problem \ Activity		Feature investigation		Feature implement-ation		Artifact investigation		Dependency investigation		Runtime interaction investigation		Artifact usage investigation		Execution pattern investigation		Quality assessment		Domain concept analysis	
Overloaded views	No overview		S		S				S								S		
		CV		CV				CV								CV			
Narrow focus on static source perspectives and views	Distributed artifacts	H	S			H	S	H	S	H	S			H	S	H	S	H	S
		CV	FE			CV	FE	CV	FE	CV	FE			CV	FE	CV	FE	CV	FE
	Collaboration hidden	H	S			H	S	H	S	H	S			H	S	H	S	H	S
		CV	FE			CV	FE	CV	FE	CV	FE			CV	FE	CV	FE	CV	FE
	Execution paths hidden					H	S	H	S	H	S	H	S	H	S				
						CV		CV		CV		CV		CV					
	Imprecise static source code					H	S			H	S			H	S				
	Features hidden in code																		
			FE		FE														FE

Figure 11.1: The various IDE shortcomings addressed by the proposals presented in the second part of the dissertation (*Hermion*, *Senseo*, *CollView*, and *FeatureEnv*) and the development activities to which these proposals contribute (H = *Hermion*, S = *Senseo*, CV = *CollView*, FE = *FeatureEnv*).

instance which other packages used a particular package and which other packages were used by this package at runtime. *CollView* exploits similar collaboration information but visualizes the revealed communication patterns between source elements in navigable charts. *FeatureEnv* eventually represents entire software features by showing which entities are used in a feature and how, that is, which method invoked which other method in which order. These four approaches combined are able to represent in the IDE collaboration between artifacts from source level up to feature level, thus covering a large spectrum of how different types of source elements dynamically communicate with each other.

Hidden execution paths. *Hermion* reveals hidden execution paths in source code by showing which code fragments are executed how often, which types are stored in variables, and which methods are invoked at specific call sites in a method. *Senseo* also makes explicit which methods are invoked in a particular method or which other methods this method invoked, but additionally shows on a class or package level how such elements are used at runtime, that is, which are their callers and which other elements they call. *CollView* visualizes in method collaboration charts the execution paths between methods for particular system executions. *CollView* also visualizes in dedicated collaboration charts how and how often classes or packages execute each other. These charts can be opened for specific artifacts directly in the conventional static source perspectives.

Unclear static source code. Static source code is often difficult to understand, for instance because it refers to abstract types, but from reading the static code it does not become obvious what kind of concrete types will be used at runtime. At polymorphic call sites it is statically not determinable which methods will be invoked during system execution. *Hermion* addresses this problem by enriching the static source code perspectives of IDEs with gathered dynamic information such as types of variables, information about the methods being invoked at runtime at specific call sites, information about the callers of a method, the types of arguments being passed during invocation, or the types of objects receiving the message send that triggered the method invocation.

Features hidden in code. Software features are intangible artifacts, features purely exist at runtime of a software system and usually encompass many static source artifacts contributing to their execution. Conventional IDEs thus do not represent features in their static source perspectives. With *FeatureEnv* we embed a feature representation in the IDE by visualizing all entities being part of a feature's execution and by displaying how these artifacts communicate with each other during feature execution, that is, how the method call tree of this feature is constructed. Thus, *FeatureEnv* supports developers in comprehending a software feature by providing a tangible representation of its implementation.

Missing Overview. *Senseo* and *CollView* partially contribute to a better system overview as they represent high-level collaboration, for instance by visualizing the communication patterns between the different system packages. Knowing how packages collaborate with each other is a starting point to identifying parts of a system being important for particular tasks or a general system understanding. Thus, these two approaches help developers to quickly gain an overview of the important parts or aspects of a software system.

11.2 Problems Previously Addressed

In the first part of this dissertation, we presented three approaches aiming at mitigating the information overload problem in IDEs, in particular to be able to enhance the IDE with dynamic information. These three proposals (*HeatMaps*, *SmartGroups*, and *AutumnLeaves*) tackle the following problems:

Information overload, missing overview. *HeatMaps* highlight artifacts being relevant for particular development tasks in the static source perspectives using a heat-coloring scheme (a color gradient from blue to red). Thus, developers can quickly identify important artifacts, hence have to deal with less information and can eventually faster gain an overview of the system as important and relevant artifacts stand out. *SmartGroups* support the automatic identification of task-relevant entities and group them together; for instance, *SmartGroups* present a group of artifacts being relevant for defect correction tasks. Developers can hence focus on a subset of all system artifacts when working on a task, which reduces the information overload. One aspect of the information overload in IDEs is the fact that a developer's workspace is usually cluttered with many open views, tabs, or windows. *AutumnLeaves* offers a mechanism to automatically identify and eventually close unused tabs or windows in the workspace. Hence, *AutumnLeaves* performs "housekeeping" in the workspace to reduce the amount of information developers have to deal with.

Missing context and task support. Both *HeatMaps* and *SmartGroups* represent task-relevant context, that is, they identify the set of entities being of importance for a specific development task. *HeatMaps* color such related artifacts with heat colors. However, this form of context representation is rather difficult to use, thus we provide with *SmartGroups* a means to identify and group entities likely to be relevant for specific types of development tasks. Developers can manipulate the proposed groups by adding or removing entities they personally consider to (not) be relevant. Thus, *SmartGroups* provide a working context to developers that helps them to stay focused on the task-at-hand.

11.3 Remaining Problems

We do not claim to have solved all problems of modern IDEs with our seven proposals presented in this dissertation. There are several shortcomings and issues of IDEs still not yet properly or completely addressed. We elaborate on the IDE issues we consider as still being pending in the following. This identification of pending issues we also recognize as an avenue of future work.

Limited support for quality assessment. As discussed in Section 1.1.2, traditional IDEs do not offer advanced support to help developers assess the quality of the software systems they are maintaining. For instance,

an IDE could specifically highlight entities highly coupled with different parts of a system or draw a developer's attention to inefficiently implemented algorithms or to artifacts never used at runtime. With *Senseo* we provide some degree of quality assessment by making visible in the IDE source artifacts that are expensive to execute, for instance because they are creating many objects or execute many bytecode instructions. Such artifacts (packages, classes, or methods) are colored in red in package tree and source code editor by *Senseo*.

However, none of our approaches particularly aims at assessing software quality. There is still much room for improvement when it comes to quality assessment support in the IDE. Software quality should be of high priority in software development, thus the IDE should encourage it, for example by (i) highlighting artifacts not passing defined quality metrics such as fan-in/fan-out [MARI 07], (ii) by observing developers while typing code and suggesting solutions to improve code quality (*e.g.* avoiding code clones), or (iii) by notifying programmers of artifacts or parts of artifacts (*e.g.* branches of if-statements) not covered by the system's test suite. In short, IDEs have a high potential to improve system quality, but existing approaches such as Lint [JOHN 78] do not exploit all possibilities of IDEs to assist developers in improving system quality during software development.

System overview. Gaining an overview of an unfamiliar system is a highly complex and time-consuming activity. Although many of our proposals contribute to improve the overview of a system in the IDE, we acknowledge that there is still more work to be done to further ease the task of getting an overview of a large software system. We identify the following points on which further work in this area could focus:

Task-dependent visualizations. When developers need to understand a software system and thus want to gain an overview of it, they usually want to accomplish a certain task, for instance they want to adapt this system to use it for a specific goal or they want to locate a particular system feature. However, aids to overview a system are usually unaware of specific goals, for instance visualizations of a system's static structure or its dynamic behavior always look the same for each task and are not tailored to specific tasks. Thus, we propose to integrate visualizations that focus on aspects of a software system that are relevant for a developer's current task and goal. For instance, if a developer wants to understand how a feature is implemented in a system, a visualization of the entire system structure should highlight the artifacts used in that particular feature.

Starting points. Many visualizations provide a "big picture" view of a system, but it is often unclear for a developer how to go from the abstract to the concrete. Even though visualizations such as the system complexity view usually highlight artifacts with respect to some criteria such as size (number of lines of code, number of methods or attributes), it is not obvious which entities developers should explore first in order to improve their understanding of the system. Highlighted artifacts are often not those really crucial for system understanding, but they might stand out since their implementation is complex. We recommend that visualizations or any other means helping developers in the understanding and overviewing process should suggest starting points, that is, entities being appropriate candidates to analyze in detail in order to boost system understanding.

Overview of low level collaboration. While *Senseo* and *CollView* give an overview of high level collaboration, for instance by visualizing the communication patterns between packages, none of our approaches provides an overview of the low level dynamic collaboration between methods or classes. *Senseo*, *CollView*, and *Hermion* integrate collaboration information in the source code views or in dedicated visualizations, but this information is locally available for selected methods or classes for which developers are presented with the direct callers or callees of a particular artifact. While these views on collaboration information can be navigated to locate indirect callers and callees, there is no overview of such low level collaboration available, for instance by means of visualizations showing the general communication patterns between methods in the system. *FeatureEnv* provides a visualization of the method call tree generated by a specific feature, but this visualization does not reveal how the involved methods communicate in other system features. Furthermore, the method call tree visualized in *FeatureEnv* often grows very large and does thus not offer a comprehensible overview of a feature's execution. We wonder whether it is possible to embed in the IDE interactive visualizations providing an overview of low level dynamic collaboration patterns between methods and classes. In order to be effective, such visualizations should show a compact view of these collaboration patterns, otherwise they are too large to be usable and understandable.

With this discussion we complete the second part of the dissertation. The last part concludes this work by summarizing the main contributions we made and by elaborating on the perspectives for further work concerned with development environments and the problems they face.

Part III

Conclusions

The last part of this dissertation concludes this work. We first present the contributions of this dissertation (Chapter 12) and subsequently elaborate on perspectives for further work in the context of development environments (Chapter 13).

Chapter 12

Contributions

We set out to alleviate two main problems of IDEs: (i) information overload and (ii) a narrow focus on static source perspectives neglecting any dynamic information of software systems. We proposed to enhance the IDE perspectives with dynamic information to make dynamic dependencies between distributed source artifacts visible. A careful analysis of the shortcomings of conventional IDEs, however, revealed that we first have to tackle the information overload issue before we are able to embed more information in the already overloaded and busy views of IDEs. This analysis of the current state of modern IDEs yielded a detailed list of shortcomings and issues (cf. Figure 1.2) for which we subsequently proposed various approaches to address them. We focused on the Squeak or Pharo Smalltalk IDE and the Eclipse Java IDE.

We also analyzed the state of the art in research on development environments and identified shortcomings of related works such as Mylyn [KERS 05, KERS 06] or NavTracks [SING 05] and proposed to amend and extend existing approaches in order to properly and more completely address the information overload in IDEs by highlighting or categorizing entities relevant for software maintenance tasks. We learnt from existing works how to analyze and visualize runtime behavior of software systems (*e.g.* [TANT 03, BIND 07, KLEY 88, DUCA 04]) and adapted, extended, and improved these works to augment IDEs with dynamic information to achieve our goal of integrating up to date and accurate dynamic information in IDEs.

The analysis of the shortcomings of IDEs and of related work in this area allowed us to propose seven distinct approaches tackling the issue of information overload or narrow focus on static source structure, or both. These seven approaches combined answer the research question of this

dissertation, namely how we can tackle the information overload in IDEs while at the same time reasonably integrate dynamic information in the static source views.

The approaches to mitigate the information overload problem are based on (i) highlighting artifacts of interest to quickly identify them in the large software space, (ii) representing context to be able to focus on specific artifacts, and (iii) reducing the number of open views in a developer's workspace. To augment static source perspectives with dynamic information, we claim that it is crucial to embed this additional dynamic information in the already existing and familiar source code views such as package tree, source editor, or source browser to not further overload the busy IDE workspace with new views and perspectives and to lower the burden for developers to adopt and use dynamic information during software maintenance. Our proposals meet this requirement and are, for instance, able to explicitly represent dynamic collaboration or software features in the IDE.

The seven approaches tackling information overload and augmenting static source perspectives with dynamic information form the foundation of the contributions of this dissertation; we briefly summarize all seven contributed proposals in the following:

Approaches alleviating information overload:

- *HeatMaps* highlight task-relevant artifacts by coloring them in a heat color scheme in accordance with the degree-of-interest for the task-at-hand. Hence, developers can more quickly navigate to important artifacts in the software space and have to deal with less information. We validated *HeatMaps* by means of a benchmark validation using recorded development sessions consisting of nearly 90'000 navigation and modification events. *HeatMaps* are available for the Squeak or Pharo Smalltalk IDE.

- *SmartGroups* represent working context in the IDE by categorizing entities relevant for specific tasks. Developers can thus focus on a small subset of all source artifacts in a system and are not anymore affected by overloaded views. *SmartGroups* were evaluated with a benchmark validation analyzing development sessions consisting of nearly 50'000 navigation and modification events stemming from five different developers working on six distinct systems. *SmartGroups* are available for the Squeak or Pharo Smalltalk IDE.

- *AutumnLeaves* performs "housekeeping services" in the overloaded IDE workspace by automatically removing unused views on source

artifacts (windows, tabs). Thus, developers have to deal with fewer windows, which increases the overview and reduces information overload. We evaluated *AutumnLeaves* with a benchmark validation considering 25 development sessions of eight different developers working in distinct software systems. *AutumnLeaves* is available for the Squeak or Pharo Smalltalk IDE and for the Eclipse Java IDE.

Proposals to augment source perspectives narrowly focusing on static structure with dynamic information:

- *Hermion* augments the static source code of the dynamically typed language Smalltalk with dynamic information such as runtime types of variables or receiver types of message sends. *Hermion* helps developers understanding static source code and execution flow therein. We evaluate this approach by analyzing developer feedback reporting about *Hermion*'s usefulness for maintaining software systems. *Hermion* is available for the Squeak and Pharo Smalltalk IDE.

- *Senseo* integrates in the source code perspectives of the Eclipse Java IDE dynamic information aggregated over several system executions to make visible dynamic collaboration between statically distributed source artifacts. Furthermore, *Senseo* provides a visualization of the calling context tree representing specific executions. We thoroughly validated *Senseo* by means of a controlled empirical experiment with 30 professional developers solving typical software maintenance tasks. This experiment reveals that *Senseo* aids developers to more efficiently (17.5% less time spent) and more correctly (33.5% improvement) solve these maintenance tasks compared to working with the conventional Eclipse IDE.

- *CollView* visualizes collaboration patterns between static source artifacts (packages, classes, methods) and thus supports developers in navigating and gaining an overview of dynamic relationships in a system. We report on developer feedback to illustrate the usefulness of *CollView*. *CollView* is available for the Squeak Smalltalk IDE.

- *FeatureEnv* explicitly represents software features in the Squeak Smalltalk IDE by providing different feature visualizations. These feature views allow developers to visually compare different features, to understand how the source artifacts interact during feature execution, and to quickly identify in the source space the artifacts being exercised by a feature. We comprehensively evaluated *FeatureEnv* by means of a controlled empirical experiment with twelve developers correcting defects in an unfamiliar software system. The

> experimental results show that developers using *FeatureEnv* can more quickly locate (56% less time spent) and correct (33% less time spent) the erroneous source artifacts responsible for the defect compared to developers using the standard Squeak Smalltalk IDE.

While these approaches have been developed individually and separately, they nicely complement each other and can be combined in the same IDE to comprehensively tackle the information overload and an IDE's narrow focus on static source perspectives.

In order to build the necessary body of evidence that these approaches indeed successfully confirm our thesis, we validated each approach either with benchmarks analyzing empirical data or with controlled empirical experiments involving professional developers as subjects. The validation of our proposals revealed (i) that they correctly and efficiently work (for instance, that *AutumnLeaves* closes windows actually not being used anymore or that *Hermion* efficiently gathers dynamic information) and (ii) that they solve or at least mitigate the problems each approach claims to address. Table 12.1 summarizes with which methods we evaluated the seven proposals and what is the outcome of each validation.

We conclude from the outcome of the various evaluations we performed (cf. Table 12.1) that the contributions of this dissertation (i) are capable of reducing the information overload in the IDE, that (ii) integrating dynamic information helps developers to more efficiently and effectively perform software maintenance tasks, and that (iii) this integrated dynamic information does not further overload an IDE's workspace provided that we tightly and seamlessly embed the dynamic information in the already existing and familiar source views and that the overload with (static) information has been alleviated upfront. Consequently, the seven contributed approaches altogether successfully substantiate our thesis that we first need to mitigate the information overload in the static views of IDEs on source code, and subsequently augment these existing and familiar views on the software structure with dynamic information to effectively use dynamic information for software maintenance tasks.

We complete this dissertation by emphasizing in the subsequent chapter interesting perspectives for further work.

Proposal	*Validation*	*Outcome*
HeatMaps	Benchmark validation, user feedback	*HeatMaps* reduce information overload and improve overview by accurately highlighting relevant entities, thus enabling developers to focus on interesting entities.
SmartGroups	Benchmark validation, user feedback	*SmartGroups* accurately categorize entities relevant for maintenance tasks to ultimately reduce information overload and to allow developers to more efficiently perform such tasks.
AutumnLeaves	Benchmark validation, user feedback	*AutumnLeaves* correctly identifies and closes unused windows and reduces information overload by "cleaning" a developer's workspace.
Hermion	Performance evaluation, user feedback	*Hermion* efficiently integrates dynamic information in the IDE; developers consider such information as useful for software maintenance.
Senseo	Controlled experiment (30 subjects), performance evaluation	*Senseo* efficiently integrates dynamic information in the IDE which enables developers to more effectively and more correctly solve maintenance tasks.
CollView	Performance evaluation, user feedback	*CollView* efficiently visualizes dynamic collaborations between source artifacts in the IDE to help developers during system maintenance.
FeatureEnv	Controlled experiment (12 subjects)	*FeatureEnv*'s representation of software features in the IDE enables developers to more efficiently correct software defects.

Table 12.1: How we validated each proposal and the outcome of these validations.

Chapter 13

Perspectives

Better system overview. *HeatMaps* marginally contribute to a better overview of the static aspects of a system by highlighting task-relevant artifacts. *CollView* improves the overview of system behavior and *FeatureEnv* of specific software features. None of these approaches, however, satisfactorily supports developers in gaining a general overview of a system, thus we also integrated in the IDE software visualizations such as the system complexity view (cf. Chapter A). Yet still we envision a better support of IDEs to gain an overview of an unfamiliar system. For instance, a visualization integrated in IDEs should be able to highlight the source elements that are important for system understanding, for example by suggesting valuable entry points to start the system comprehending process in order to conceive a system in its entireness.

Improve overview of low level collaboration. Having an overview of dynamic behavior on a method level is not well supported by our proposals. While *Hermion* and *Senseo* integrate information about method execution in an IDE's source perspectives, they do not contribute a means providing an overview of such information. *CollView* and *FeatureEnv* visualize how methods invoke each other but focus on specific methods or features. We envision to support developers in understanding general method invocation and execution flow patterns in a system, for instance by integrating compact, lightweight visualizations of method call trees aggregated over several system executions to give a general impression of how methods communicate with each other.

Task-dependent overview. *HeatMaps* and *SmartGroups* acknowledge the fact that system investigation activities are usually concerned with particular software maintenance tasks. For instance, a developer seeks to locate system artifacts containing a particular defect. Means to gain an overview of a system such as visualizations, however, do not account for the task to be accomplished. We thus aim at providing task-sensitive visualizations that enable developers to quickly overview the part of a system relevant for the task-at-hand, for instance the artifacts and their relationships causing a defect.

Supporting quality assessment. Even though conventional IDEs just provide limited support for quality assessment, we have not addressed this shortcoming in this dissertation. As software quality is a crucial aspect of software development, we plan to help developers identifying quality problems in the software systems they are maintaining by extending the IDE to automatically run quality metrics and report on any violations of these metrics, such as source artifacts being badly covered with tests or artifacts that are highly coupled with many distinct parts of the system. *Senseo* has already started to support quality assessment by highlighting artifacts creating many objects or executing many bytecode instructions.

Practicality of the approaches. For many approaches such as *HeatMaps*, *SmartGroups*, or *AutumnLeaves* we provide a thorough analysis of their accuracy and correctness, that is, whether they correctly achieve what they are supposed to achieve. As we aim at supporting developers in performing software maintenance tasks, it is, however, crucial to also assess the practical usefulness of our work, that is, whether our proposals are usable in practice and whether developers can indeed benefit from their availability in the IDE. Such a validation has to be performed by controlled empirical experiments (like those performed for *Senseo* and *FeatureEnv*) or by field studies (such as presented for Mylyn [KERS 06]) to gather reliable quantitative and qualitative feedback on the practicality of our work.

Comprehensive validation of combined approaches. So far, we validated each approach individually to reveal its correctness or practical usefulness. It would be, however, interesting to find out how much all approaches combined contribute to more efficiently and effectively accomplish software maintenance tasks in the IDE. To this end, we aim at conducting controlled experiments similar to the one performed for *Senseo*, but providing the experimental subject group with several, pos-

sibly all approaches we contributed in this dissertation at the same time in the IDE. Such an experiment could measure the combined effect of our techniques and answer questions like whether such a combination contributes more than the sum of the individual parts.

Supporting more IDEs. The contributed approaches are either available for the Squeak and Pharo Smalltalk IDE or for the Eclipse Java IDE. *AutumnLeaves* is available for all three IDEs. We are currently working on porting, adapting, and enhancing some approaches to also make them available for other IDEs, for instance by implementing *FeatureEnv* in the Eclipse IDE. We also started to work on Visual Studio [MICR 10] for C# to enhance this IDE with the concept of working context similar as provided by *SmartGroups*.

Part IV

Appendices

Appendix A

Additional IDE Enhancements

The first part of this dissertation discussed several approaches to mitigate the information overload in IDEs. Our proposals such as *HeatMaps*, *SmartGroups*, or *AutumnLeaves*, however, do not completely solve the information overload problem. In particular, gaining a quick, higher-level overview of the system is still not easily possible. To tackle the problem of higher-level overview, we integrated several visualizations in the IDE such as a system complexity view, class blueprints, and UML diagrams. Another enhancement to the IDE makes use of icons placed next to source artifacts to show information not directly visible in the static software structure, such as whether a method is overridden in subclasses or whether a message has any senders. This section briefly discusses all these additional techniques we implemented.

A.1 Visualizations

Visualizations presenting results from software analyses are often provided by dedicated environments such as Moose [DUCA 00], Program Explorer [LANG 95], GraphTrace [KLEY 88], or CodeCrawler [LANZ 05]. Thus these visualizations are usually not integrated in the IDE, that is, in the environment where developers mostly work on software systems. As these reverse-engineering environments are not focusing on providing means to modify the system under study, developers interested in obtaining software visualizations have to use both the IDE and the reverse-

engineering tool to work on their daily maintenance tasks. Developers are forced to frequently switch between the two environments. In order to avoid such tool and context switches and to be able to use software analysis knowledge while actually working with source code, we integrated useful software analysis visualizations in the IDE. The visualizations are accessible from the conventional source perspectives.

We extended the perspectives on static source artifacts (*e.g.* classes) in the Squeak and Pharo Smalltalk IDE with appropriate visualizations, similar to what RBCrawler [KUHN 07] and inCode [INCO 09] do for Cincom Smalltalk [VISU 10] and for Eclipse [ECLI 03], respectively. When the developer has selected a class, she can generate directly within the IDE a class blueprint, a system complexity view focusing on the class hierarchy of this particular class, or a UML class diagram instead of just looking at the source code. Typically, a visualization is interactive in the sense that the developer can click on nodes to jump to classes or methods represented by such nodes. Context menus accessible from within visualizations allow the developer to trigger searches for references to classes, for implementors of message sends, or for accessors of instance variables.

In the following we look in detail at three types of visualizations we integrated in the Squeak and Pharo IDE, especially at the particular IDE problems each visualization type mitigates.

A.1.1 System Complexity View

To integrate the system complexity view into the OmniBrowser framework [BERG 07b] implementing the various browsers used in the Squeak or Pharo Smalltalk IDE, we added a tab bar next to the source code view to switch from the traditional source code editor to the visualization.

One kind of visualization available in this tab bar is the system complexity view [LANZ 03]. Usually the system complexity view visualizes the class hierarchies contained in the currently selected package. The view can be extended with additional packages. Furthermore, the system complexity view can also be generated just for a set of specific classes. Classes not included in the set of currently selected classes or packages but being part of a class hierarchy whose root class is included, are depicted in the system complexity view with a green border. The system complexity view is a polymetric view, which means it maps several metrics to each shape representing a class. The width of this shape represents number of attributes, the height number of methods, and the color number of lines of code of a particular class.

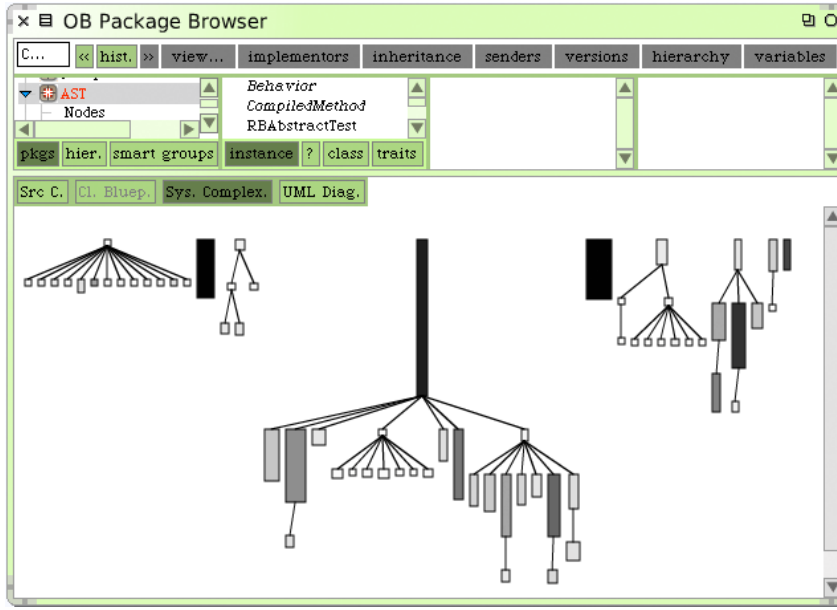

Figure A.1: System complexity view of the AST package integrated in the Squeak OmniBrowser IDE.

Such a view serves as a navigational aid and helps developers to gain a quick overview of all classes in a system or of a particular package and thus mitigates the information overload problem by providing a higher view on source entities than the traditional source perspectives such as the package explorer. Figure A.1 shows an example for a system complexity view showing a package modeling the abstract syntax tree (AST) of Smalltalk.

A.1.2 Class Blueprint

Class blueprints [DUCA 05b] are similarly integrated in the OmniBrowser IDE as the system complexity view. For a selected class, the blueprint shows in different layers its methods and attributes as nodes and the communication between them as edges. Five layers are shown: the initialization, interface, implementation, accessor, and attributes layer. Each method of the class belongs to one of the four method layers while all its attributes are included in the attributes layer. To determine the communication between the methods and attributes, the source code is statically analyzed. The class blueprint is also a polymetric view: The

width of a method node represents the number of statically defined invocations, the height its lines of code. The color of a method node encodes further information, such as whether the method is an abstract or a delegator method.

As static analysis might yield imprecise results, for instance proposing communication edges that are never triggered at runtime, we extended the class blueprint to also take into account dynamic information recorded while executing particular software features. A method invocation or an attribute access actually occurring in a recorded execution scenario is denoted as a red edge while the edges for statically determined communication between methods appear in black, those for accessing attributes in cyan. Such dynamic class blueprints support developers in understanding the class-internal execution patterns occurring during specific software features. As the dynamic information can be aggregated for many different software runs, a dynamic class blueprint is also useful to detect dead code, *i.e.* methods never invoked or attributes never accessed.

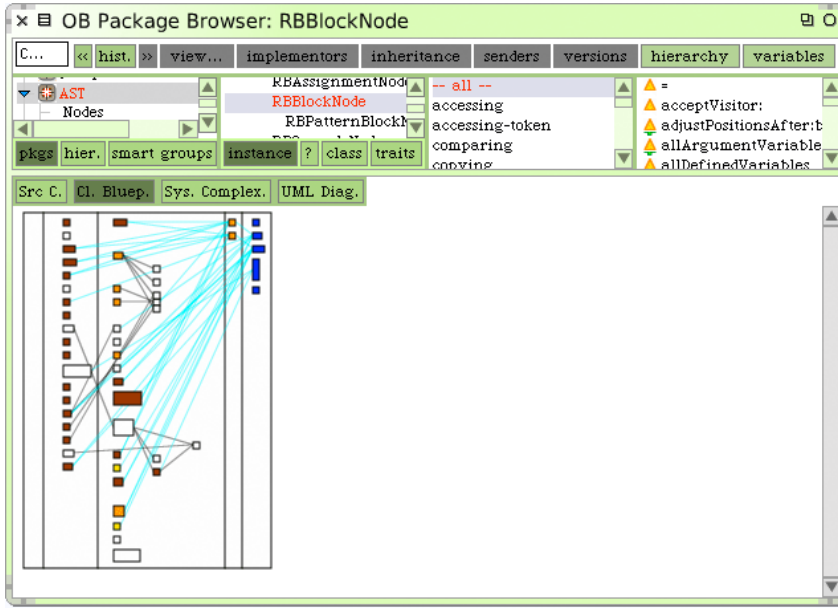

Figure A.2: Class blueprint of the RBBlockNode class.

Developers can interact with the class blueprint, for instance to navigate to the source code of a method, to see a list of all class-internal callers of a method or a list of all methods referencing an attribute. From the traditional source code views, developers can open the blueprint of the currently selected class. The currently selected method is highlighted in

the class blueprint for better visibility. Figure A.2 shows the blueprint of the class representing the block node in the AST; in this case, the internal communication is determined by static analysis only.

A.1.3 UML Class Diagrams

As a third visualization, we added support for drawing UML class diagrams to the OmniBrowser IDE. Similar as for the system complexity view, we can show UML diagrams for the selected package, class, or a set of classes. The class diagram shows all attributes and methods of a class and connects classes based on inheritance relationships. Class-side methods are underlined in the diagram.

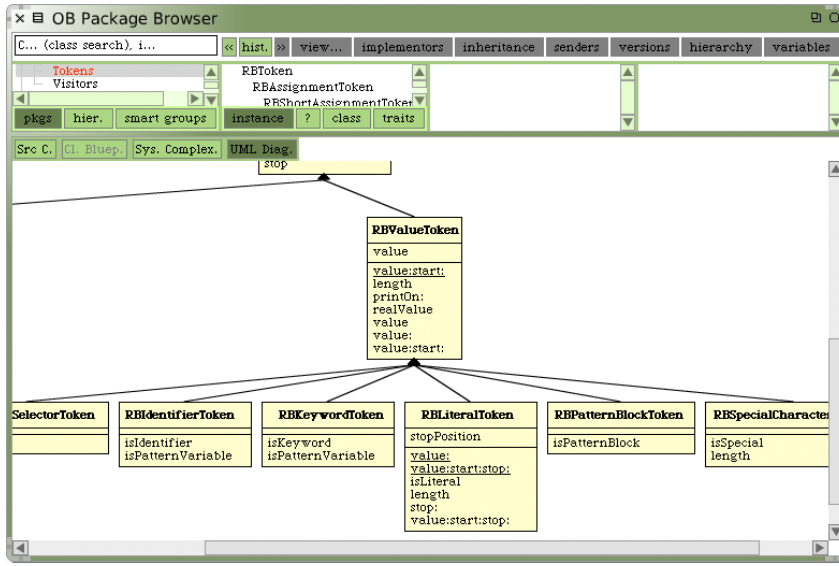

Figure A.3: UML class diagram of a part of the AST package.

The UML class diagram is interactive as developers can click on methods to browse to their source code or on attributes to see a list of methods referencing them. Clicking on a class' title bar to see its definition in source is also possible. The integration of UML class diagrams in the IDE supports developers in gaining an overview of classes and their hierarchies. Such class diagrams provide a more detailed view than the system complexity view as they also show all methods and attributes of each class. Figure A.3 shows a part of the same AST package as Figure A.1, but in a UML class diagram.

A.2 Iconic Information

Icons serve the purpose of visually conveying information that is otherwise difficult to grasp or display [LEWI 04], such as information about inheritance relationships, for instance whether a method is overridden in subclasses. As the user interface in an IDE is usually already overloaded with information, small, non-intrusive icons serve well the purpose of giving a quick hint about interesting aspects of source artifacts. Thus, well-designed icons are a suitable means to enrich purely textual interfaces. Another advantage of icons is that they do not use much space: a twelve by twelve pixel icon already conveys valuable information.

However, it is crucial to not overuse icons. They should only be used to draw the attention of developers to important information; if they are used everywhere in all IDE interfaces, they cannot serve the purpose of highlighting information of interest. Furthermore, a concise set of icons should be integrated; developers will not be able to remember or to interpret their respective meaning if too many, just slightly different icons are used. Additionally, the icons need to be self-explanatory. If these requirements are met, icons support developers to more quickly grasp important information in a large software space and thus help them to deal with the information overload problem, in particular as icons also make explicit information that can otherwise not easily be spotted in the source space, such as whether a method is overridden in any subclass or whether it is raising an exception.

We integrated icons in the Squeak and Pharo Smalltalk IDE. We opted to not show more than one icon per source artifact at a time. Thus, if an artifact fulfills the criteria of several icons, we only show the one with the highest precedence. Precedence is determined by the criticality of the information depicted with the icon. Errors, for instance, have the highest precedence followed by important structural information such as being an abstract method or one that is overridden in subclasses. We show icons for all types of source entities used in Smalltalk: packages, classes, methods, and, as a special case, test methods and classes. Table A.1 reports on the different icons used for the four main types of source artifacts while Figure A.4 shows the various method icons appearing when browsing the String class.

We also use icons as a means to navigate in the source space. Clicking on icons triggers the execution of an action appropriate for this particular icon. The overridden icon, for instance, can be used to navigate to the method in a subclass overriding the selected method. If several subclasses override this method, then the developer sees a list of classes to choose from. Some icons trigger non-navigational actions when clicked. Test-

Table A.1: Icons available in Squeak Smalltalk for different source artifacts.

Name	*Icon*	*Description*
Package, Class Categories		
Package icon		Denotes whether an entity is a Monticello package.
Dirty package		Packages that have been locally modified but not yet committed.
Newer version		Packages with newer version(s) in repository than installed locally.
Classes		
Exception icon		Exception and subclasses.
Collection icon		Collection and subclasses.
Methods		
Overridden icon		Whether a method is overridden in any subclass.
Overrides icon		Whether a method overrides the same method from a superclass.
Overrides and overridden icon		Whether a method overrides and is overridden at the same time.
Super send icon		Method sending super to the same method.
Super send icon		Super send, but invoking different receiver.
Abstract icon		Abstract method, that is, one sending #isSubclassResponsibility.
Halt icon		Method sending #halt.
Flag icon		Method sending #flag:.
Exception icon		Method raising an exception.
Test methods, test classes		
Green icon		Test method or class running green.
Yellow icon		Test method or class running yellow (failures).
Red icon		Test method or class running red (errors).
More green than red icon		Test class with more green than red running test methods.
Equal green than red icon		Test class with nearly the same number of green and red running test methods.
More red than green icon		Test class with more red than green running test methods.
Not run icon		Test has not been executed yet.

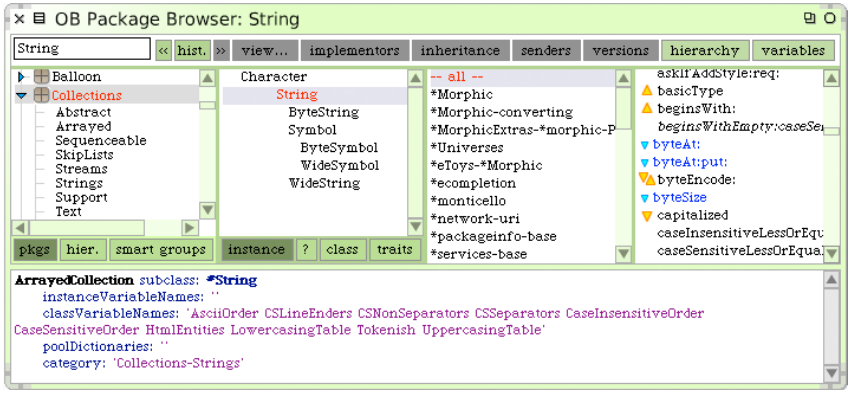

Figure A.4: Several icons appear when browsing class String, such as the abstract, overridden, overrides, or overrides and overridden icon.

related icons for example run the associated test class or method when clicked. Hence icons do not only show additional information, but can also provide useful facilities to navigate in the source space or serve as shortcuts for specific actions.

Bibliography

[AMMO 97] G. Ammons, T. Ball, and J. R. Larus. *Exploiting hardware performance counters with flow and context sensitive profiling*. In PLDI '97: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, pp 85–96. ACM Press, 1997. (pp 174, 178, 182)

[ARIS 07] E. Arisholm, H. Gallis, T. Dyba, and D. I. Sjoberg. *Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise*. IEEE Transactions on Software Engineering, vol. 33, no. 2, pp 65–86, 2007. (p 186)

[ARNO 01] M. Arnold and B. G. Ryder. *A Framework for Reducing the Cost of Instrumented Code*. In SIGPLAN Conference on Programming Language Design and Implementation, pp 168–179, 2001. (p 57)

[BALL 99] T. Ball. *The Concept of Dynamic Analysis*. In Proceedings of the European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC'99), number 1687 in LNCS, pp 216–234, Heidelberg, sep 1999. Springer Verlag. (pp 55, 165, 216)

[BASI 97] V. Basili. *Evolving and Packaging Reading Technologies*. Journal Systems and Software, vol. 38, no. 1, pp 3–12, 1997. (pp 94, 121, 224)

[BERG 07a] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. *Stateful Traits*. In Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006), volume 4406 of *LNCS*, pp 66–90. Springer, August 2007. (p 108)

[BERG 07b] A. Bergel, S. Ducasse, C. Putney, and R. Wuyts. *Meta-Driven Browsers*. In Advances in Smalltalk — Proceedings of 14th

International Smalltalk Conference (ISC 2006), volume 4406 of *LNCS*, pp 134–156. Springer, August 2007. (pp 162, 272)

[BIND 07] W. Binder, J. Hulaas, and P. Moret. *Advanced Java Bytecode Instrumentation*. In PPPJ'07: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, pp 135–144, New York, NY, USA, 2007. ACM Press. (pp 56, 176, 259)

[BLAC 06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. *The DaCapo Benchmarks: Java Benchmarking Development and Analysis*. In OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, pp 169–190, New York, NY, USA, October 2006. ACM Press. (p 194)

[BLAC 09] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. Pharo by Example. Square Bracket Associates, 2009. (p 1)

[BRAD 92] K. Brade, M. Guzdial, M. Steckel, and E. Soloway. *Whorf: A Visualization Tool for Software Maintenance*. In Proceedings of IEEE Workshop on Visual Languages, pp 148–154. IEEE Society Press, 1992. (pp 52, 245)

[BRAN 98] J. Brant, B. Foote, R. Johnson, and D. Roberts. *Wrappers to the Rescue*. In Proceedings European Conference on Object Oriented Programming (ECOOP'98), volume 1445 of *LNCS*, pp 396–417. Springer-Verlag, 1998. (p 56)

[BRIA 06] L. C. Briand, Y. Labiche, and J. Leduc. *Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software*. IEEE Transactions on Software Engineering, vol. 32, no. 9, pp 642–663, 2006. (p 206)

[BYKO 08] V. Bykov. *Hopscotch: Towards User Interface Composition*. In International Workshop on Advanced Software Development Tools and Techniques (WasDeTT), July 2008. (pp 24, 28)

[CAME 96] D. Cameron, B. Rosenblatt, and E. Raymond. Learning GNU Emacs. O'Reilly, 1996. (p 2)

[CHU- 03] M. C. Chu-Carroll, J. Wright, and A. T. T. Ying. *Visual separation of concerns through multidimensional program storage*. In AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development, pp 188–197, New York, NY, USA, 2003. ACM Press. (pp 92, 94)

[CONS 92] M. P. Consens, A. O. Mendelzon, and A. G. Ryman. *Visualizing and Querying Software Structures*. In Proceedings of the 14th International Conference on Software Engineering, pp 138–156, 1992. (p 48)

[CORB 89] T. A. Corbi. *Program Understanding: Challenge for the 1990's*. IBM Systems Journal, vol. 28, no. 2, pp 294–306, 1989. (pp 94, 121, 224)

[CORN 07a] B. Cornelissen. *Dynamic Analysis Techniques for the Reconstruction of Architectural Views*. In Proceeding of the 14th Working Conference on Reverse Engineering (WCRE). IEEE, 2007. (p 158)

[CORN 07b] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. *Understanding Execution Traces Using Massive Sequence and Circular Bundle Views*. In Proceedings of the 15th International Conference on Program Comprehension (ICPC), pp 49–58. IEEE Computer Society, 2007. (p 218)

[CORN 09] B. Cornelissen, A. Zaidman, A. van Deursen, and B. van Rompaey. *Trace Visualization for Program Comprehension: A Controlled Experiment*. In Proceedings 17th International Conference on Program Comprehension (ICPC), pp 100–109. IEEE Computer Society, 2009. (p 184)

[CUBR 03] D. Cubranic and G. Murphy. *Hipikat: Recommending Pertinent Software Development Artifacts*. In Proceedings 25th International Conference on Software Engineering (ICSE 2003), pp 408–418, New York NY, 2003. ACM Press. (pp 33, 34, 41, 198)

[DE A 08] B. de Alwis and G. C. Murphy. *Answering conceptual queries with Ferret*. In Proceedings of the 30th International Conference on Software Engineering (ICSE), pp 21–30, New York, NY, USA, 2008. ACM. (pp 4, 46, 47, 48, 167, 197)

[DE P 93] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. *Visualizing the Behavior of Object-Oriented Systems*. In Proceedings

of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93), pp 326–337, October 1993. (pp 58, 71, 198, 219, 245)

[DeLi 05a] R. DeLine, M. Czerwinski, and G. G. Robertson. *Easing Program Comprehension by Sharing Navigation Data*. In VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, pp 241–248, Washington, DC, USA, 2005. IEEE Computer Society. (pp 38, 41)

[DeLi 05b] R. DeLine. *Staying Oriented with Software Terrain Maps*. In Proceedings of the 2005 International Workshop on Visual Languages and Computing, pp 309–314. IEEE Computer Society, 2005. (p 52)

[Deme 00] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Finding Refactorings via Change Metrics*. In Proceedings of 15th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00), pp 166–178, New York NY, 2000. ACM Press. Also in ACM SIGPLAN Notices 35 (10). (p 151)

[Deme 03] S. Demeyer, S. Ducasse, K. Mens, A. Trifu, and R. Vasa. *Report of the ECOOP'03 Workshop on Object-Oriented Reengineering*. In Object-Oriented Technology (ECOOP'03 Workshop Reader), LNCS, pp 72–85. Springer-Verlag, 2003. (pp 1, 150, 224)

[Denk 06] M. Denker, O. Greevy, and M. Lanza. *Higher Abstractions for Dynamic Analysis*. In 2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006), pp 32–38, 2006. (p 159)

[Denk 07] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. *Sub-Method Reflection*. In Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007, volume 6/9, pp 231–251. ETH, October 2007. (pp 56, 76, 159, 220)

[Desm 06] M. Desmond, M.-A. Storey, and C. Exton. *Fluid Source Code Views*. In ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06), pp 260–263, Washington, DC, USA, 2006. IEEE Computer Society. (pp 2, 5, 27, 31, 167, 197)

[Dmit 04a] M. Dmitriev. *Profiling Java Applications Using Code Hotswapping and Dynamic Call Graph Revelation*. In Proceedings of

the Fourth International Workshop on Software and Performance, pp 139–150. ACM Press, 2004. (p 7)

[DMIT 04b] M. Dmitriev. *Profiling Java Applications Using Code Hotswapping and Dynamic Call Graph Revelation*. In WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance, pp 139–150. ACM Press, 2004. (pp 44, 45, 57, 176, 197)

[DUCA 00] S. Ducasse, M. Lanza, and S. Tichelaar. *Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems*. In Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools), June 2000. (p 271)

[DUCA 04] S. Ducasse, M. Lanza, and R. Bertuli. *High-Level Polymetric Views of Condensed Run-Time Information*. In Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04), pp 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press. (pp 51, 58, 208, 219, 259)

[DUCA 05a] S. Ducasse, M. Lanza, and R. Robbes. *Multi-level Method Understanding Using Microprints*. In Proceedings of VISSOFT 2005 (3th IEEE International Workshop on Visualizing Software for Understanding), September 2005. (pp 26, 31, 60)

[DUCA 05b] S. Ducasse and M. Lanza. *The Class Blueprint: Visually Supporting the Understanding of Classes*. Transactions on Software Engineering (TSE), vol. 31, no. 1, pp 75–90, January 2005. (pp 51, 273)

[DUCA 06] S. Ducasse, T. Gîrba, and R. Wuyts. *Object-Oriented Legacy System Trace-based Logic Testing*. In Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06), pp 35–44. IEEE Computer Society Press, 2006. (p 56)

[DUFO 03a] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. *Dynamic metrics for Java*. ACM SIGPLAN Notices, vol. 38, no. 11, pp 149–168, November 2003. (pp 57, 176)

[DUFO 03b] B. Dufour, L. Hendren, and C. Verbrugge. **J: A tool for dynamic analysis of Java programs*. In OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp 306–307, New York, NY, USA, 2003. ACM Press. (p 57)

[DUNS 00] A. Dunsmore, M. Roper, and M. Wood. *Object-Oriented Inspection in the Face of Delocalisation*. In Proceedings of ICSE '00 (22nd International Conference on Software Engineering), pp 467–476. ACM Press, 2000. (pp 1, 55, 68, 92, 94, 120, 150, 151, 226)

[ECLI 03] Eclipse. *Eclipse Platform: Technical Overview*, 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf. (pp 1, 152, 157, 166, 203, 205, 218, 272)

[EICK 92] S. G. Eick, J. L. Steffen, and S. Eric E., Jr. *SeeSoft—A Tool for Visualizing Line Oriented Software Statistics*. IEEE Transactions on Software Engineering, vol. 18, no. 11, pp 957–968, November 1992. Depth. (pp 25, 26, 31, 54, 60, 84, 137)

[EISE 03] T. Eisenbarth, R. Koschke, and D. Simon. *Locating Features in Source Code*. IEEE Computer, vol. 29, no. 3, pp 210–224, March 2003. (pp 8, 225, 245)

[EISE 05] A. Eisenberg and K. De Volder. *Dynamic Feature Traces: Finding Features in Unfamiliar code*. In Proceedings IEEE International Conference on Software Maintenance (ICSM 2004), pp 337–346, Los Alamitos CA, September 2005. IEEE Computer Society Press. (p 8)

[FITT 54] P. M. Fitts. *The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement*. Journal of Experimental Psychology, vol. 47, no. 6, pp 381–391, 1954. (p 124)

[FLAJ 90] P. Flajolet, P. Sipala, and J.-M. Steyaert. *Analytic variations on the common subexpression problem*. In Automata, Languages, and Programming, volume 443 of *LNCS*, pp 220–234. Springer Verlag, 1990. (p 231)

[FURN 86] G. W. Furnas. *Generalized Fisheye View*. In Proceedings of CHI '86 (Conference on Human Factors in Computing Systems), pp 16–23. ACM Press, 1986. (p 51)

[GAMM 93] E. Gamma, R. Helm, J. Vlissides, and R. E. Johnson. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. In O. Nierstrasz, editor, Proceedings ECOOP '93, volume 707 of *LNCS*, pp 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag. (p 204)

[GAMM 95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Mass., 1995. (p 202)

[Gî 04] T. Gîrba, S. Ducasse, and M. Lanza. *Yesterday's Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes*. In Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), pp 40–49, Los Alamitos CA, September 2004. IEEE Computer Society. (pp 70, 71)

[GOLD 84] A. Goldberg. Smalltalk 80: the Interactive Programming Environment. Addison Wesley, Reading, Mass., 1984. (p 122)

[GOTH 05] G. Goth. *Beware the March of This IDE: Eclipse Is Overshadowing Other Tool Technologies*. IEEE Software, vol. 22, no. 4, pp 108–111, 2005. (p 2)

[GRAV 00] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. *Predicting Fault Incidence Using Software Change History*. IEEE Transactions on Software Engineering, vol. 26, no. 2, 2000. (pp 99, 101)

[GREE 06] O. Greevy, S. Ducasse, and T. Gîrba. *Analyzing Software Evolution through Feature Views*. Journal of Software Maintenance and Evolution: Research and Practice (JSME), vol. 18, no. 6, pp 425–456, 2006. (p 8)

[GREE 07] O. Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Bern, May 2007. (p 228)

[HAMO 03] A. Hamou-Lhadj and T. Lethbridge. *An Efficient Algorithm for Detecting Patterns in Traces of Procedure Calls*. In Proceedings of 1st International Workshop on Dynamic Analysis (WODA), May 2003. (pp 209, 216, 231, 245)

[HAMO 04] A. Hamou-Lhadj and T. Lethbridge. *A Survey of Trace Exploration Tools and Techniques*. In Proceedings IBM Centers for Advanced Studies Conferences (CASON 2004), pp 42–55, Indianapolis IN, 2004. IBM Press. (p 55)

[HAMO 05] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. *Recovering Behavioral Design Models from Execution Traces*. In Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005), pp 112–121, Los Alamitos CA, 2005. IEEE Computer Society Press. (pp 1, 55)

[HASS 04] A. Hassan and R. Holt. *Predicting Change Propagation in Software Systems*. In Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04), pp 284–293,

Los Alamitos CA, September 2004. IEEE Computer Society Press. (p 70)

[INCO 09] inCode. *inCode — Eclipse plugin for code analysis*, 2009. http://www.intooitus.com/inCode.html. (pp 205, 272)

[INGA 97] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. *Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself*. In Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97), pp 318–326. ACM Press, November 1997. (pp 1, 5, 205, 211, 218, 224, 225, 226, 228)

[JANZ 03] D. Janzen and K. de Volder. *Navigating and Querying Code Without Getting Lost*. In AOSD'03: Proceedings of the 2nd International Conference on Aspect-oriented Software Development, pp 178–187, New York, NY, USA, 2003. ACM. (pp 45, 48)

[JERD 96] D. Jerding, J. Stasko, and T. Ball. *Visualizing Message Patterns in Object-Oriented Program Executions*. Research Report GIT-GVU-96-15, Georgia Institute of Technology, May 1996. (pp 8, 71, 226)

[JOHN 78] S. Johnson. *Lint, a C Program Checker*. In UNIX programmer's manual, pp 78–1273. AT&T Bell Laboratories, 1978. (p 253)

[JONE 04] J. A. Jones, A. Orso, and M. J. Harrold. *GAMMATELLA: visualizing program-execution data for deployed software*. Information Visualization, vol. 3, no. 3, pp 173–188, 2004. (p 59)

[JUNK 09] M. Junker. Kumpel: Visual Exploration of File Histories. Master's thesis, University of Bern, January 2009. (p 53)

[KANJ 99] G. K. Kanji. 100 Statistical Tests. SAGE Publications, 1999. (p 239)

[KAZM 99] R. Kazman and S. J. Carriere. *Playing detective: Reconstructing software architecture from available evidence.* Automated Software Engineering, April 1999. (p 51)

[KERS 05] M. Kersten and G. C. Murphy. *Mylar: a degree-of-interest model for IDEs*. In AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, pp 159–168, New York, NY, USA, 2005. ACM Press. (pp 2, 4, 5, 39, 41, 68, 69, 80, 84, 94, 97, 168, 198, 259)

[KERS 06] M. Kersten and G. C. Murphy. *Using task context to improve programmer productivity*. In SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, pp 1–11, New York, NY, USA, 2006. ACM Press. (pp 2, 39, 41, 97, 168, 198, 259, 266)

[KICZ 97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-Oriented Programming*. In M. Akşit and S. Matsuoka, editors, Proceedings of European Conference on Object-Oriented Programming, volume 1241, pp 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997. (p 176)

[KICZ 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. *An overview of AspectJ*. In J. L. Knudsen, editor, Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP-2001), volume 2072 of *Lecture Notes in Computer Science*, pp 327–353, 2001. (pp 56, 176)

[KLEY 88] M. F. Kleyn and P. C. Gingrich. *GraphTrace — Understanding Object-Oriented Systems using Concurrently Animated Views*. In Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'88), volume 23, pp 191–205. ACM Press, November 1988. (pp 58, 59, 198, 219, 245, 246, 259, 271)

[KO 04] A. J. Ko and B. A. Myers. *Designing the whyline: a debugging interface for asking questions about program behavior*. In Proceedings of the 2004 conference on Human factors in computing systems, pp 151–158. ACM Press, 2004. (pp 7, 41, 42, 45)

[KO 05] A. J. Ko, H. Aung, and B. A. Myers. *Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks*. In ICSE '05: Proceedings of the 27th international conference on Software engineering, pp 125–135, 2005. (pp 92, 94, 121)

[KUHN 07] A. Kuhn. *RBCrawler — a Visual Navigation System for Smalltalk's Refactoring Browser*. European Smalltalk User Group Innovation Technology Award, August 2007. (p 272)

[KUHN 08] A. Kuhn, P. Loretan, and O. Nierstrasz. *Consistent Layout for Thematic Software Maps*. In Proceedings of 15th Working Conference on Reverse Engineering (WCRE'08), pp 209–218,

Los Alamitos CA, October 2008. IEEE Computer Society Press. (p 52)

[LANG 95] D. Lange and Y. Nakamura. *Interactive Visualization of Design Patterns can help in Framework Understanding*. In Proceedings ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95), pp 342–357, New York NY, 1995. ACM Press. (pp 58, 219, 245, 246, 271)

[LANZ 01] M. Lanza. *The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques*. In Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution), pp 37–42, 2001. (p 71)

[LANZ 03] M. Lanza and S. Ducasse. *Polymetric Views—A Lightweight Visual Approach to Reverse Engineering*. Transactions on Software Engineering (TSE), vol. 29, no. 9, pp 782–795, September 2003. (pp 51, 272)

[LANZ 05] M. Lanza and S. Ducasse. *CodeCrawler — An Extensible and Language Independent 2D and 3D Software Visualization Tool*. In Tools for Software Maintenance and Reengineering, RCOST / Software Technology Series, pp 74–94. Franco Angeli, Milano, 2005. (p 271)

[LEWI 04] J. Lewis, R. Rosenholtz, N. Fong, and U. Neumann. *VisualIDs: automatic distinctive icons for desktop interfaces*. ACM Transactions on Graphics, vol. 23, no. 3, pp 416–423, August 2004. (p 276)

[LICA 03] D. Licata, C. Harris, and S. Krishnamurthi. *The Feature Signatures of Evolving Programs*. In Proceedings IEEE International Conference on Automated Software Engineering, pp 281–285, Los Alamitos CA, October 2003. IEEE Computer Society Press. (pp 227, 230, 236)

[LIEN 09] A. Lienhard, J. Fierz, and O. Nierstrasz. *Flow-Centric, Back-In-Time Debugging*. In Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2009, volume 33 of *LNBIP*, pp 272–288. Springer-Verlag, 2009. (pp 43, 45, 167, 197)

[LIKE 32] R. Likert. *A technique for the measurement of attitudes*. Archives of Psychology, vol. 22, no. 140, pp 1–55, 1932. (pp 164, 185, 215)

[LLOY 82] S. P. LLoyd. *Least Squares Quantization in PCM*. IEEE Transactions on Information Theory, vol. 28, pp 129–137, 1982. (p 180)

[LÖWE 01] W. Löwe, A. Ludwig, and A. Schwind. *Understanding Software – Static and Dynamic Aspects*. In 17th International Conference on Advanced Science and Technology, pp 52–57, 2001. (pp 55, 151, 166)

[MALE 02] J. I. Maletic, A. Marcus, and M. Collard. *A Task Oriented View of Software Visualization*. In Proceedings of the 1st Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2002), pp 32–40. IEEE, June 2002. (p 51)

[MALO 04] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick. *Scratch: A Sneak Preview*. In International Conference on Creating, Connecting and Collaborating through Computing, pp 104–109. IEEE Computer Society, 2004. (p 42)

[MARC 04] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. *An Information Retrieval Approach to Concept Location in Source Code*. In Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004), pp 214–223, November 2004. (pp 93, 94)

[MARI 07] M. Marin, A. v. Deursen, and L. Moonen. *Identifying crosscutting concerns using fan-in analysis*. ACM Transactions on Software Engineering and Methodology, vol. 17, no. 1, pp 1–37, 2007. (p 253)

[MEHT 02] A. Mehta and G. Heineman. *Evolving legacy systems features using regression test cases and components*. In Proceedings ACM International Workshop on Principles of Software Evolution, pp 190–193, New York NY, 2002. ACM Press. (p 224)

[MEND 95] A. Mendelzon and J. Sametinger. *Reverse Engineering by Visualizing and Querying*. Software — Concepts and Tools, vol. 16, pp 170–182, 1995. (p 51)

[MEYE 06] M. Meyer, T. Gîrba, and M. Lungu. *Mondrian: An Agile Visualization Framework*. In ACM Symposium on Software Visualization (SoftVis'06), pp 135–144, New York, NY, USA, 2006. ACM Press. (p 205)

[MICR 10] Microsoft. *Microsoft Visual Studio*, March 2010. http://www.microsoft.com/VisualStudio. (pp 1, 267)

[MILL 56] J. C. Miller and C. J. Maloney. *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*. Psychological Review, vol. 63, pp 81–97, 1956. (p 6)

[MORE 09] P. Moret, W. Binder, D. Ansaloni, and A. Villazón. *Visualizing Calling Context Profiles with Ring Charts*. In VISSOFT 2009: 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, pp 33–36, Edmonton, Alberta, Canada, September 2009. IEEE Computer Society. (pp 174, 181)

[MURP 06] G. C. Murphy, M. Kersten, and L. Findlater. *How are Java software developers using the Eclipse IDE?* IEEE Software, jul 2006. (p 7)

[NETB 10] NetBeans. *NetBeans IDE*. http://www.netbeans.org, archived at http://www.webcitation.org/5p1qB6hNt, 2010. (p 1)

[NIEL 89a] F. Nielson. *The Typed Lambda-Calculus with First-Class Processes*. In E. Odijk and J.-C. Syre, editors, Proceedings PARLE '89, Vol II, volume 366 of *LNCS*, pp 357–373, Eindhoven, June 1989. Springer-Verlag. (pp 1, 94)

[NIEL 89b] J. Nielsen and J. T. Richards. *The Experience of Learning and Using Smalltalk*. IEEE Software, vol. 6, no. 3, pp 73–77, 1989. (pp 1, 224)

[NIER 05] O. Nierstrasz, S. Ducasse, and T. Gîrba. *The Story of Moose: an Agile Reengineering Environment*. In Proceedings of the European Software Engineering Conference (ESEC/FSE'05), pp 1–10, New York NY, 2005. ACM Press. Invited paper. (pp 51, 53)

[O'BR 05] M. O'Brien, J. Buckley, and C. Exton. *Empirically Studying Software Practitioners - Bridging the Gap between Theory and Practice*. In Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005). IEEE Computer Society Press, 2005. (p 242)

[PACI 04] M. Pacione, M. Roper, and M. Wood. *A Novel Software visualisation Model to Support Software Comprehension*. In Proceedings of the 11th Working Conference on Reverse Engineering, pp 70–79. IEEE Computer Society, November 2004. (pp 2, 3, 186, 193, 194)

[PARN 06] C. Parnin and C. Görg. *Building Usage Contexts During Program Comprehension*. In Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06), volume 0, pp 13–22, Los Alamitos CA, 2006. IEEE Computer Society. (pp 68, 81)

[PINZ 05] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. *Visualizing Multiple Evolution Metrics*. In Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization), pp 67–75, St. Louis, Missouri, USA, May 2005. (p 71)

[POTH 07] G. Pothier, E. Tanter, and J. Piquer. *Scalable Omniscient Debugging*. Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07), vol. 42, no. 10, pp 535–552, 2007. (pp 7, 43)

[RAPI 98] P. Rapicault, M. Blay-Fornarino, S. Ducasse, and A.-M. Dery. *Dynamic Type Inference to Support Object-Oriented Reengineering in Smalltalk*, 1998. Proceedings of the ECOOP '98 International Workshop Experiences in Object-Oriented Reengineering, abstract in Object-Oriented Technology (ECOOP '98 Workshop Reader forthcoming LNCS). (p 156)

[RECH 07] J. Rech and W. Schäfer. *Visual support of software engineers during development and maintenance*. SIGSOFT Softw. Eng. Notes, vol. 32, no. 2, pp 1–3, 2007. (p 29)

[REIS 03] S. P. Reiss. *Visualizing Java in Action*. In Proceedings of SoftVis 2003 (ACM Symposium on Software Visualization), pp 57–66, 2003. (pp 58, 166, 198, 219, 245, 246)

[REIS 05] S. P. Reiss. *JOVE: Java as it happens*. In Proceedings of SoftVis 2005(ACM Symposium on Software Visualization), pp 115–124, 2005. (pp 58, 198, 219, 245)

[RENG 06] L. Renggli. Magritte — Meta-Described Web Application Development. Master's thesis, University of Bern, June 2006. (pp 213, 226, 230, 235)

[RENG 07] L. Renggli. *Pier — The Meta-Described Content Management System*. European Smalltalk User Group Innovation Technology Award, August 2007. Won the 3rd prize. (p 161)

[RICH 02] T. Richner and S. Ducasse. *Using Dynamic Information for the Iterative Recovery of Collaborations and Roles*. In Proceedings

of 18th IEEE International Conference on Software Maintenance (ICSM'02), p 34, Los Alamitos CA, October 2002. IEEE Computer Society. (pp 59, 198, 218)

[ROBB 05] R. Robbes, S. Ducasse, and M. Lanza. *Microprints: A Pixel-based Semantically Rich Visualization of Methods*. In Proceedings of 13th International Smalltalk Conference (ISC'05), pp 131–157, 2005. (p 54)

[ROBB 08] R. Robbes and M. Lanza. *How Program History Can Improve Code Completion*. In Proceedings of ASE 2008 (23rd International Conference on Automated Software Engineering), pp 317–326, 2008. (pp 77, 109, 130)

[ROBI 03a] M. P. Robillard and G. C. Murphy. *FEAT: A tool for locating, describing, and analyzing concerns in source code*. In Proceedings of 25th International Conference on Software Engineering, pp 822–823, May 2003. (pp 35, 41, 84, 96, 198)

[ROBI 03b] M. P. Robillard and G. C. Murphy. *Automatically inferring concern code from program investigation activities*. In Proceedings of the 18th International Conference on Automated Software Engineering, pp 225–234, October 2003. (pp 35, 36, 97)

[ROBI 07] M. P. Robillard and G. C. Murphy. *Representing Concerns in Source Code*. ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 16, no. 1, p 3, 2007. (p 35)

[RÖTH 07a] D. Röthlisberger, O. Greevy, and O. Nierstrasz. *Feature Driven Browsing*. In Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), pp 79–100. ACM Digital Library, 2007. (p 20)

[RÖTH 07b] D. Röthlisberger, O. Greevy, and A. Lienhard. *Feature-centric Environment*. In Proceedings IEEE International Workshop on Visualizing Software for Understanding (Vissoft 2007) (tool demonstration), 2007. (p 20)

[RÖTH 07c] D. Röthlisberger and O. Nierstrasz. *Combining Development Environments with Reverse Engineering*. In Proceedings of FAMOOSr 2007 (Ist International Workshop on FAMIX and Moose in Reengineering), 2007. (p 20)

[RÖTH 07d] D. Röthlisberger, M. Denker, and É. Tanter. *Unanticipated Partial Behavioral Reflection*. In Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006), volume 4406 of *LNCS*, pp 47–65. Springer, 2007. (p 159)

[RÖTH 08a] D. Röthlisberger, O. Greevy, and O. Nierstrasz. *Exploiting Runtime Information in the IDE*. In Proceedings of the 16th International Conference on Program Comprehension (ICPC 2008), pp 63–72, Los Alamitos, CA, USA, 2008. IEEE Computer Society. (pp 7, 19, 71, 76)

[RÖTH 08b] D. Röthlisberger. *Hermion — Exploiting the Dynamics of Software*. European Smalltalk User Group Innovation Technology Award, August 2008. (p 19)

[RÖTH 08c] D. Röthlisberger and O. Greevy. *Representing and Integrating Dynamic Collaborations in IDEs*. In Proceedings of the 15th Working Conference on Reverse Engineering (WCRE 2008), pp 74–78, Los Alamitos, CA, USA, 2008. IEEE Computer Society. (p 20)

[RÖTH 09a] D. Röthlisberger, O. Nierstrasz, and S. Ducasse. *Autumn Leaves: Curing the Window Plague in IDEs*. In Proceedings of the 16th Working Conference on Reverse Engineering (WCRE 2009), pp 237–246, Los Alamitos, CA, USA, 2009. IEEE Computer Society. (pp 5, 6, 19, 108)

[RÖTH 09b] D. Röthlisberger, O. Nierstrasz, S. Ducasse, and A. Bergel. *Tackling Software Navigation Issues of the Smalltalk IDE*. In Proceedings of International Workshop on Smalltalk Technologies (IWST 2009), pp 58–67, New York, NY, USA, 2009. ACM. (pp 5, 7, 19)

[RÖTH 09c] D. Röthlisberger, O. Nierstrasz, S. Ducasse, D. Pollet, and R. Robbes. *Supporting Task-oriented Navigation in IDEs with Configurable HeatMaps*. In Proceedings of the 17th International Conference on Program Comprehension (ICPC 2009), pp 253–257, Los Alamitos, CA, USA, 2009. IEEE Computer Society. (pp 19, 39, 180, 181)

[RÖTH 09d] D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. *Augmenting Static Source Views in IDEs with Dynamic Metrics*. In Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009), pp 253–262, Los Alamitos, CA, USA, 2009. IEEE Computer Society. (pp 19, 174, 184)

[RÖTH 09e] D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. *Senseo: Enriching Eclipse's Static Source Views with Dynamic Metrics*. In Proceedings of the 25th International Conference on Software

Maintenance (ICSM 2009), pp 383–384, Los Alamitos, CA, USA, 2009. IEEE Computer Society. Tool demo. (pp 19, 174)

[ROUN 03] A. Rountev, A. Milanova, and B. G. Ryder. *Fragment Class Analysis for Testing of Polymorphism in Java Software*. In ICSE '03: Proceedings of the 25th IEEE International Conference on Software Engineering, pp 210–220, Los Alamitos, CA, USA, 2003. IEEE Computer Society Press. (p 7)

[ROUN 04] A. Rountev, S. Kagan, and M. Gibas. *Evaluating the imprecision of static analysis*. In PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp 14–16, New York, NY, USA, 2004. ACM. (p 7)

[SHIR 03] J. S. Shirabad, T. C. Lethbridge, and S. Matwin. *Mining the Maintenance History of a Legacy Software System*. In International Conference on Software Maintenance (ICSM 2003), pp 95–104, 2003. (pp 32, 34)

[SING 05] J. Singer, R. Elves, and M.-A. Storey. *NavTracks: Supporting Navigation in Software Maintenance*. In International Conference on Software Maintenance (ICSM'05), pp 325–335, Washington, DC, USA, sep 2005. IEEE Computer Society. (pp 1, 2, 4, 5, 36, 41, 68, 69, 71, 80, 84, 86, 87, 97, 168, 198, 259)

[SOLO 86] E. Soloway and K. Ehrlich. *Empirical studies of programming knowledge*. Readings in artificial intelligence and software engineering, pp 507–521, 1986. (p 94)

[SQUE 10] Squeak. *Squeak Home Page*. http://www.squeak.org/, archived at http://www.webcitation.org/5p1poT9Ta, 2010. (pp 157, 161, 166)

[STAS 90] J. T. Stasko. *TANGO: A Framework and System for Algorithm Animation*. IEEE Computer, vol. 23, no. 9, pp 27–39, September 1990. (p 51)

[STAS 98] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price. Software Visualization — Programming as a Multimedia Experience. The MIT Press, 1998. (p 50)

[STAS 00] J. Stasko. *An evaluation of space-filling information visualizations for depicting hierarchical structures*. Int. J. Hum.-Comput. Stud., vol. 53, no. 5, pp 663–694, 2000. (p 181)

[STOR 01] M.-A. Storey, C. Best, and J. Michaud. *SHriMP Views: An Interactive and Customizable Environment for Software Exploration*. In Proceedings of International Workshop on Program Comprehension (IWPC '2001), 2001. (p 52)

[SUN 00] Sun Microsystems, Inc. *Java Virtual Machine Profiler Interface (JVMPI)*. Web pages at http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/, 2000. (p 57)

[SYST 99] T. Systä. *On the relationships between static and dynamic models in reverse engineering Java software*. In Working Conference on Reverse Engineering (WCRE99), pp 304–313, October 1999. (p 151)

[SYST 01] T. Systä, K. Koskimies, and H. Müller. *Shimba — An Environment for Reverse Engineering Java Software Systems*. Software — Practice and Experience, vol. 31, no. 4, pp 371–394, January 2001. (pp 59, 198, 219)

[TAEN 89] D. Taenzer, M. Ganti, and S. Podar. *Problems in Object-Oriented Software Reuse*. In S. Cook, editor, Proceedings ECOOP '89, pp 25–38, Nottingham, July 1989. Cambridge University Press. (p 8)

[TANT 03] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. *Partial Behavioral Reflection: Spatial and Temporal Selection of Reification*. In Proceedings of OOPSLA '03, ACM SIGPLAN Notices, pp 27–46, nov 2003. (pp 56, 106, 158, 159, 218, 259)

[TARV 09] A. Tarvo. *Mining Software History to Improve Software Maintenance Quality: A Case Study*. IEEE Software, vol. 26, no. 1, pp 34–40, January 2009. (pp 70, 99)

[TILL 94] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. *Programmable Reverse Enginnering*. International Journal of Software Engineering and Knowledge Engineering, vol. 4, no. 4, pp 501–520, 1994. (p 51)

[VILL 08] A. Villazón, W. Binder, and P. Moret. *Aspect Weaving in Standard Java Class Libraries*. In PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, pp 159–167, New York, NY, USA, September 2008. ACM. (pp 176, 182)

[VILL 09] A. Villazón, W. Binder, and P. Moret. *Flexible Calling Context Reification for Aspect-Oriented Programming*. In AOSD '09:

Proceedings of the 8th International Conference on Aspect-oriented Software Development, pp 63–74, Charlottesville, Virginia, USA, March 2009. ACM. (pp 176, 182, 183)

[VISU 10] VisualWorks. *Cincom Smalltalk*. http://www.cincomsmalltalk.com/, archived at http://www.webcitation.org/5p1rRxls5, 2010. (pp 1, 224, 272)

[VOK 04] M. Vok. *Defect Frequency and Design Patterns: An Empirical Study of Industrial Code*. IEEE Transactions on Software Engineering, vol. 30, pp 904–917, 2004. (p 205)

[WALK 00] R. J. Walker, G. C. Murphy, J. Steinbok, and M. P. Robillard. *Efficient mapping of software system traces to architectural views*. In CASCON '00: Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research, p 12. IBM Press, 2000. (p 71)

[WEIS 81] M. Weiser. *Program slicing*. In ICSE '81: Proceedings of the 5th international conference on Software engineering, pp 439–449, Piscataway, NJ, USA, 1981. IEEE Press. (p 166)

[WETT 08] R. Wettel and M. Lanza. *Visual Exploration of Large-Scale System Evolution*. In Proceedings of Softvis 2008 (4th International ACM Symposium on Software Visualization), pp 155 – 164. IEEE CS Press, 2008. (p 53)

[WILC 45] F. Wilcoxon. Individual Comparisons by Ranking Methods. International Biometric Society, 1945. (p 241)

[WILD 92] N. Wilde and R. Huitt. *Maintenance Support for Object-Oriented Programs*. IEEE Transactions on Software Engineering, vol. SE-18, no. 12, pp 1038–1044, December 1992. (pp 1, 7, 8, 55, 68, 92, 120, 150, 151, 224)

[WILD 93] N. Wilde, P. Matthews, and R. Hutt. *Maintaining Object-Oriented Software*. IEEE Software (Special Issue on "Making O-O Work"), vol. 10, no. 1, pp 75–80, January 1993. (p 202)

[WILD 95] N. Wilde and M. Scully. *Software Reconnaisance: Mapping Program Features to Code*. Journal on Software Maintenance: Research and Practice, vol. 7, no. 1, pp 49–62, 1995. (pp 8, 245)

[XIE 06] X. Xie, D. Poshyvanyk, and A. Marcus. *Visualization of CVS Repository Information*. In WCRE'06: Proceedings of the 13th

Working Conference on Reverse Engineering, pp 231–242, Washington, DC, USA, 2006. IEEE Computer Society. (p 34)

[YING 04] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. *Predicting Source Code Changes by Mining Change History*. Transactions on Software Engineering, vol. 30, no. 9, pp 573–586, 2004. (p 32)

[ZAID 05] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. *Applying Webmining Techniques to Execution Traces to Support the Program Comprehension Process*. In Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR'05), pp 134–142, Los Alamitos CA, 2005. IEEE Computer Society Press. (p 55)

[ZELL 02] A. Zeller. *Isolating cause-effect chains from computer programs*. In SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, pp 1–10, New York, NY, USA, 2002. ACM Press. (p 244)

[ZELL 03] A. Zeller. *Program analysis: A hierarchy*. In Proceedings of the ICSE 2003 Workshop on Dynamic Analysis, pp 6–9, 2003. (p 166)

[ZIMM 04a] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. *Mining Version Histories to Guide Software Changes*. In 26th International Conference on Software Engineering (ICSE 2004), pp 563–572, Los Alamitos CA, 2004. IEEE Computer Society Press. (pp 32, 34, 41, 198)

[ZIMM 04b] T. Zimmermann and P. Weißgerber. *Preprocessing CVS Data for Fine-Grained Analysis*. In Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004), pp 2–6, Los Alamitos CA, 2004. IEEE Computer Society Press. (pp 70, 71)

[ZIMM 05] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. *Mining Version Histories to Guide Software Changes*. IEEE Transactions on Software Engineering, vol. 31, no. 6, pp 429–445, June 2005. (p 112)

Curriculum Vitae

Personal Information

Name	David Röthlisberger
Date of Birth	October 1, 1982
Place of Birth	Köniz, Switzerland
Nationality	Swiss

Education

2007 – 2010	Ph.D. in Computer Science at the Software Composition Group, University of Bern, Switzerland Thesis title: *Augmenting IDEs with Runtime and Development Information for Software Maintenance*
2008 – 2010	Bachelor in Business Administration, University of Bern, Switzerland
2004 – 2006	Master in Computer Science at the Software Composition Group, University of Bern, Switzerland Thesis title: *Geppetto: Enhancing Smalltalk's Reflective Capabilities with Unanticipated Reflection*
2001 – 2004	Undergraduate Degree in Computer Science at the University of Bern, Switzerland. Minors in Mathematics and Business Administration.

Complete Curriculum Vitae:
http://www.droethlisberger.ch/media/cv-davidroethlisberger.pdf